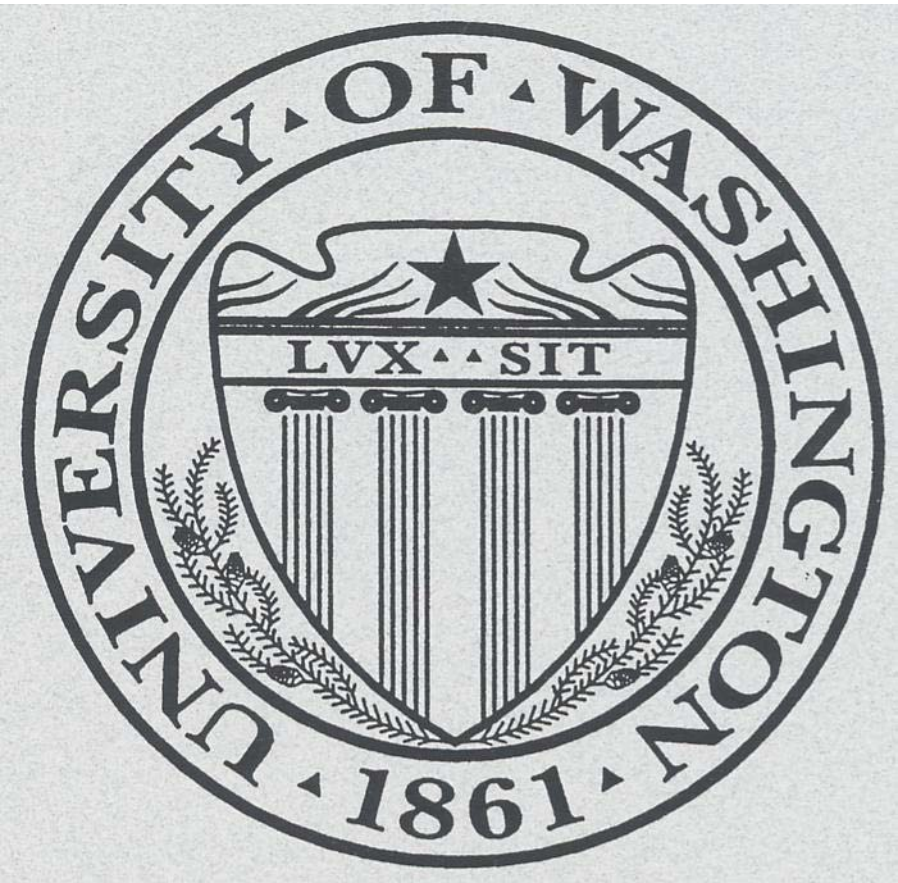
UNIVERSITY OF WASHINGTON
Department of Aeronautics and Astronautics

Numerical Techniques of Partial Differential Equations



Prepared for Professor Nathan Kutz

by

Christopher Lum
Dec. 3, 2003

Table of Contents

Abstract 3
Introduction 3
Acknowledgments 3
Nomenclature 4
Theory 5
    Finite Difference Discretization 5
    Elliptic Solve Using Fast Fourier Transform 11
    Elliptic Solve Using A\b 12
    Elliptic Solve Using LU Decomposition 12
    Elliptic Solve Using GMRES, BICGSTAB 13
    Vorticity Coefficient 14
Implementation 16
    Control Flow 16
    Computation Difficulties 18
Results 23
    Fast Fourier Transform 23
    A\b 27
    LU Decomposition 28
    BICGSTAB 29
    GMRES 32
    Spectral Implementation 35
Conclusion 62

Appendix A: MatLab Functions 63
Appendix B: MatLab Code 65
    main.m 65
    user_preferences.m 75
    create_matrices.m 79
    create_vorticity.m 80
    advection_fft.m 83
    analyze_wave_function.m 85
    ask_question.m 85
    display_parameters.m 87
    view_movie.m 88
    view_subplots.m 90
    vorticity_coefficient.m 92
Appendix C: Derivation of Equations 93
    Equations of Motion 93
    Finite Difference Discretization 96
    Elliptic Solve Using Fast Fourier Transform 102
Appendix D: Code User's Manual 104
Appendix E: Problem Statement 105

Abstract

In many applications of science and engineering, it becomes necessary to solve partial differential equations of varying complexity. Even for simpler systems, many of these partial differential equations do not have analytical solutions. In these situations it becomes necessary to rely on numerical techniques to determine the evolution of the system. This report investigates several methods to discretize and solve these partial differential equations by writing them as a large system of ordinary differential equations.

Introduction

Fluid flow is one of the most complex applications to partial differential equations. Equations that are derived from conservation of mass, momentum, and energy are often very complex and require very advanced numerical techniques to solve. Even simplified versions of these equations still require numerical techniques to solve. However, the benefits of being able to simulate fluid flow are invaluable. This report analyzes a simple evolution of vortices in a shallow fluid over time. Although this is a fairly simple simulation, it serves as a basis for more complex computational fluid dynamic algorithms.

Acknowledgments

The author would like to thank many of the people that contributed to the results contained in this report. First and foremost, the guidance of Professor Nathan Kutz of the University of Washington was invaluable. Many hours were spent in his class and office getting the code to work. In addition, many conversations with classmates and colleagues shed light on many confusing subjects.

Nomenclature

Table 1: Nomenclature used in code and in report

Report Name	MatLab Name	Description
A	A	matrix representation of Laplacian operator. ∇^2
B	B	matrix representation of partial derivative w.r.t x
C	C	matrix representation of partial derivative w.r.t y
C_ω		vorticity coefficient
∇		$\nabla = \partial/\partial x + \partial/\partial y$
M_L		number of points in x direction used for vorticity coefficient
m	m	number of discretizations in x and y direction.
N_L		number of points in y direction used for vorticity coefficient
n	n	$n = m^2$.
n		number of derivatives. ie. $\partial^n / \partial x^n$
O		of order(). ie. $O\left(n^2\right)$
k		wave number for Fourier transform
k_x	KX	wave number in x direction
k_y	KY	wave number if y direction
L	Lmatrix	lower triangular matrix decomposition of A
L	L	computational domain (20)
P	P	permutation matrix from LU decomposition.
ψ	psi	streamfunction
$\tilde{\tilde{\psi}}$	psi_t	streamfunction transformed in both x and y direction
$\overline{\psi}$	psi_col	discretized streamfunction represented as a column vector
ψ_{xx}		$\partial^2\psi / \partial x^2$. Similar notation for derivatives in different directions.
$t_{required}$		time required to solve elliptic equation
U	Umatrix	upper triangular matrix decomposition of A
υ	v	diffusion coefficient (0.001)
ω	w	vorticity
$\overline{\omega}$	w_col	discretized vorticity represented as a column vector
$\tilde{\tilde{\omega}}$	w_t	vorticity transformed in both x and y direction
ω_o	wo	initial vorticity
w.r.t		with respect to
X_L		Length of computational domain in x direction (equal to L here)
x		x direction
y		y direction
Y_L		length of computational domain in y direction (equal to L here)

Theory[1]

Presented here is a short summary of the general theory that governs this problem. For a complete derivation of these equations and the procedure, please see Appendix C: Derivation of Equations.

Finite Difference Discretization

The two governing equations of motion for this system are known as the elliptic equation and the advection diffusion equation. These are given by Equation 1 and Equation 2, respectively.

$$\nabla^2 \psi = \omega$$

Equation 1

$$\frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega + \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y}$$

Equation 2

As can be seen, these are partial differential equations in both the x and y direction. Also, they are of varying order (the Laplacian is the sum of 2nd derivatives). These are continuous operators and therefore, in order to numerically solve this problem, these continuous operators must be discretized. Recall that the second order discretization of a second derivative of a function is given by

$$\frac{d^2 f(t)}{dt^2} = \frac{f(t+\Delta t) - 2f(t) + f(t-\Delta t)}{\Delta t^2}$$

Equation 3

For convenience, we'll choose $\Delta x = \Delta y = \delta$ and for a given t, denote

$$\psi_{mn} = \psi(m, n, t_o)$$

[1] Reference [1]: Kutz, Nathan. PhD. University of Washington. Seattle, WA. Class notes for AMATH581: Practical Scientific Computing. Autumn 2003.

With these simplifications, the elliptic equation (Equation 1) can be discretized to the following form.

$$\frac{-4\psi_{m,n} + \psi_{m-1,n} + \psi_{m+1,n} + \psi_{m,n+1} + \psi_{m,n-1}}{\delta^2} = \omega_{m,n}$$

Equation 4

From Equation 4, it can be seen that the vorticity at any point in the discretization depends on the stream function at 4 neighboring points. Furthermore, with the periodic boundary conditions

$$\psi_{N+1,n} = \psi_{1,n}$$
$$\psi_{m,N+1} = \psi_{m,1}$$

A similar procedure can be performed for discretizing the partial derivative of a function in the x direction and the partial derivative of a function in the y direction. This is shown below (recall that $\Delta x = \Delta y = \delta$).

$$\frac{\partial \psi_{m,n}}{\partial x} = \frac{\psi_{m+1,n} - \psi_{m-1,n}}{2\delta}$$

Equation 5

$$\frac{\partial \psi_{m,n}}{\partial y} = \frac{\psi_{m,n+1} - \psi_{m,n-1}}{2\delta}$$

Equation 6

A simple stencil of this pattern for N = 4 is shown below in Figure 1

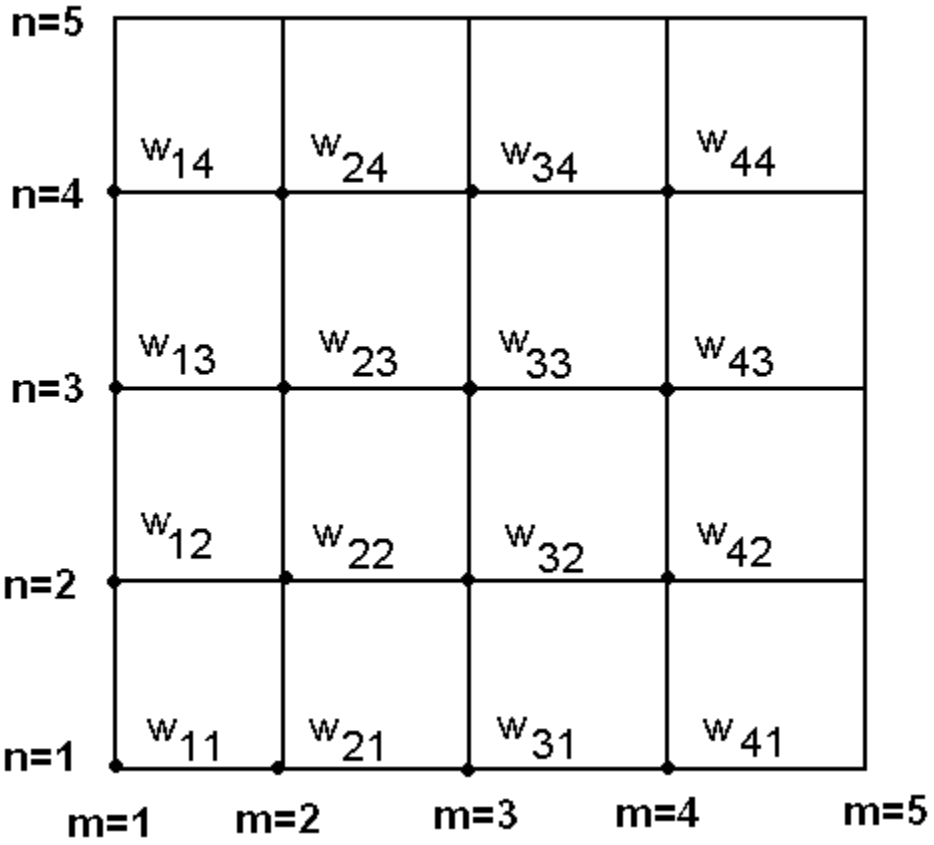


Figure 1: Pattern for vorticity and stream function at discrete points

2 matrices containing the x positions and the y positions can be computed using the MESHGRID command

[X,Y] = meshgrid(x,y)

The initial vorticity over this grid can be generated using the X and Y matrices and a given function.

$$\omega_o = f\left(X, Y\right)$$

The important thing to notice is that the resulting matrix, ω_o, will have values of vorticity at the different points stored in the following fashion.

$$\omega_o = \begin{pmatrix} \omega_{11} & \omega_{21} & \omega_{31} & \omega_{41} \\ \omega_{12} & \omega_{22} & \omega_{32} & \omega_{42} \\ \omega_{13} & \omega_{23} & \omega_{33} & \omega_{43} \\ \omega_{14} & \omega_{24} & \omega_{34} & \omega_{44} \end{pmatrix}$$

By comparing this with the stencil shown in Figure 1, one can realize that MatLab stores values of vorticity that are reflected over the horizontal middle line. That is, the value of vorticity in the upper left corner of the matrix ω_o is actually the value of vorticity in the bottom left corner of the stencil. Similarly, the value of vorticity in the upper right corner of the matrix ω_o is actually the value of vorticity in the bottom right corner of the stencil. This is important because when the reshape command is used to turn this matrix into a vector, the resulting column vector will be of the form.

wo_col = reshape(wo,16,1)

$$\omega_{o\,col} = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \omega_{14} & \omega_{21} & \omega_{22} & \omega_{23} & \omega_{24} & \omega_{31} & \omega_{32} & \omega_{33} & \omega_{34} & \omega_{41} & \omega_{42} & \omega_{43} & \omega_{44} \end{pmatrix}^T$$

Therefore, when creating the discretization of the Laplacian operator (using Equation 4) and the partial derivatives (using Equation 5 and Equation 6), the matrix must be constructed such that it is compliant with this ordering of points. The Laplacian can written in matrix form with this ordering is shown below as

$$\frac{1}{\delta^2}\begin{pmatrix}
-4 & 1 & & 1 & 1 & & & & & & & 1 & & & & \\
1 & -4 & 1 & & & 1 & & & & & & & 1 & & & \\
 & 1 & -4 & 1 & & & 1 & & & & & & & 1 & & \\
1 & & 1 & -4 & 1 & & & 1 & & & & & & & 1 & \\
1 & & & 1 & -4 & 1 & & 1 & 1 & & & & & & & \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & & & & & \\
 & & & 1 & 1 & & 1 & -4 & 1 & & & 1 & & & & \\
 & & & & 1 & & & 1 & -4 & 1 & & 1 & 1 & & & \\
 & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & \\
 & & & & & & 1 & & & 1 & -4 & 1 & & & 1 & \\
 & & & & & & & 1 & 1 & & 1 & -4 & 1 & & & 1 \\
1 & & & & & & & & 1 & & & 1 & -4 & 1 & & 1 \\
 & 1 & & & & & & & & 1 & & & 1 & -4 & 1 & \\
 & & 1 & & & & & & & & 1 & & & 1 & -4 & 1 \\
 & & & 1 & & & & & & & & 1 & 1 & & 1 & -4
\end{pmatrix}\begin{pmatrix} \psi_{11} \\ \psi_{12} \\ \psi_{13} \\ \psi_{14} \\ \psi_{21} \\ \psi_{22} \\ \psi_{23} \\ \psi_{24} \\ \psi_{31} \\ \psi_{32} \\ \psi_{33} \\ \psi_{34} \\ \psi_{41} \\ \psi_{42} \\ \psi_{43} \\ \psi_{44} \end{pmatrix} = \begin{pmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{13} \\ \omega_{14} \\ \omega_{21} \\ \omega_{22} \\ \omega_{23} \\ \omega_{24} \\ \omega_{31} \\ \omega_{32} \\ \omega_{33} \\ \omega_{34} \\ \omega_{41} \\ \omega_{42} \\ \omega_{43} \\ \omega_{44} \end{pmatrix}$$

Equation 7

As can be seen, this is a sparse matrix with only 9 diagonals that result in anything other than a zero entry. The large, sparse matrix can be referred to as the A matrix, which performs the Laplacian operation on the vector representing the stream

function. This now becomes a large matrix equation of the form $A\overline{\psi} = \overline{\omega}$. That is, given a vorticity in column form, one can use a multitude of method to solve this for the stream function vector.

The same procedure can be used to write the partial derivative in the x direction in terms of a matrix equation. By comparing Equation 4 (Laplacian) with Equation 5 (partial w.r.t. x), one can see that the equation for the partial derivative is a simplified version of the Laplacian operator. Therefore, the resulting matrix for the partial derivative w.r.t x should be the same as the A matrix with less diagonal elements and some with –1 instead of 1, and it should be divided by 2δ instead of δ^2. The two matrix operations which represent the partial w.r.t x and y, respectively, are shown below as Equation 8 and Equation 9, respectively.

$$\frac{1}{2\delta}\begin{pmatrix}
 & 1 & & & & & & & -1 & & & & & & & \\
 & & 1 & & & & & & & -1 & & & & & & \\
 & & & 1 & & & & & & & -1 & & & & & \\
 & & & & 1 & & & & & & & -1 & & & & \\
-1 & & & & & 1 & & & & & & & & & & \\
 & -1 & & & & & 1 & & & & & & & & & \\
 & & -1 & & & & & 1 & & & & & & & & \\
 & & & -1 & & & & & 1 & & & & & & & \\
 & & & & -1 & & & & & 1 & & & & & & \\
 & & & & & -1 & & & & & 1 & & & & & \\
 & & & & & & -1 & & & & & 1 & & & & \\
 & & & & & & & -1 & & & & & 1 & & & \\
1 & & & & & & & & -1 & & & & & & & \\
 & 1 & & & & & & & & -1 & & & & & & \\
 & & 1 & & & & & & & & -1 & & & & & \\
 & & & 1 & & & & & & & & -1 & & & &
\end{pmatrix}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix} = \frac{\partial}{\partial x}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix}$$

Equation 8

$$\frac{1}{2\delta}\begin{pmatrix}
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0
\end{pmatrix}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix} = \frac{\partial}{\partial y}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix}$$

Equation 9

The large, sparse matrices can be called B and C, which perform the operation of partial derivative w.r.t x and partial derivative w.r.t y, respectively.

Using these matrices, the governing equation of motion for the evolution of the fluid vorticity and stream function, Equation 1and Equation 2, can be rewritten as

$$A\bar{\psi} = \bar{\omega}$$

Equation 10

$$\frac{\partial \bar{\omega}}{\partial t} = C\bar{\psi}B\bar{\omega} - B\bar{\psi}C\bar{\omega} + \upsilon A\bar{\omega}$$

Equation 11

A multitude of methods can be used to solve the elliptic problem (Equation 10) for the stream function given a vorticity and a standard 2nd or 4th stepping scheme (ode23 or ode45) can be used to advance the advection-diffusion equation (Equation 11) into the future. Once the new value of vorticity is found, the stream function can be updated and the process can be repeated.

Elliptic Solve Using Fast Fourier Transform

Once the vorticity is known, the elliptic equation must be solved for the stream function. This can be done using a Fourier Transform very efficiently. Recall that a Fourier transform is defined as

$$F(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-ikx} f(x)\, dx$$

Equation 12

The power of the Fourier transform lies in its relation between the function and derivatives of itself. Namely,

$$\hat{F}^{(n)} = (ik)^n \hat{F}$$

Equation 13

Equation 13 shows that the Fourier transform of a derivative of power n of a function is simply the Fourier transform of the original function multiplied by a factor of $(ik)^n$, where n is the number of derivatives. For this problem, the 2-dimensional Fourier transform must be applied since there are derivatives in both the x and y directions. We can rewrite the elliptic equation as

$$\psi_{xx} + \psi_{yy} = \omega$$

By denoting the Fourier transform in x as ^ and the Fourier transform in y as ~, the elliptic equation can first be transformed in both x and y to obtain. Note that this does not affect the derivative in y.

$$-k_x{}^2 \tilde{\hat{\psi}} - k_y{}^2 \tilde{\hat{\psi}} = \tilde{\hat{\omega}}$$

This can now be solved for the stream function which is transformed in both the x and y direction. This yields

$$\tilde{\hat{\psi}} = \frac{-\tilde{\hat{\omega}}}{k_x{}^2 + k_y{}^2}$$

Equation 14

This expression can then be transformed back into the spatial domain using the inverse 2-dimensional Fourier transform.

This method is desirable since it allows for a solution of the stream function in $O\left(n\log\left(n\right)\right)$ operations.

Elliptic Solve Using A\b

In contrast to the Fast Fourier method to solve the elliptic equation, the Gaussian elimination method (A\b) is one of the slowest methods possible (2nd only to computing the inverse of A). This functions on the premise that the A matrix can be placed in row reduced echelon format. Once this is accomplished, a back substitution routine can be performed to solve for each element of the x vector. The operational cost is outlined below for this method[2]

1. Movement down N + 1 pivots
2. For each pivot, perform N additions/subtractions across a given row.
3. For each pivot, perform the addition/subtraction down the N rows.

This gives an operational cost of $O\left(n^3\right)$ to place the matrix in row reduced echelon form. In order to obtain the solution, a back substitution routine must be implemented. This has an operational cost of $O\left(n^2\right)$ and therefore, the entire scheme requires $O\left(n^3\right)+O\left(n^2\right)$ operations.

Elliptic Solve Using LU Decomposition

The method using LU decomposition functions on the assumption that the matrix A can be decomposed to into the product of two matrices, L and U, where L is in lower triangular form and U is in upper triangular form. If this is possible, then the elliptic equation can be written as

$$A\bar{\psi} = \bar{\omega}$$

$$LU\bar{\psi} = \bar{\omega} \qquad \underline{\text{let:}}\ \bar{y} = U\bar{\psi}$$

$$L\bar{y} = \bar{\omega}$$

[2] Page 41 Reference [1]

The vector $\bar{y}$ can then be solved using

$$\bar{y} = L \setminus \bar{\omega}$$

Equation 15

Once the vector $\bar{y}$, the stream function can be solved using another \ command.

$$\bar{\psi} = U \setminus \bar{y}$$

Equation 16

The natural question to ask at this point is given that this takes two \ commands instead of simply one in the A\b technique, will this actually take twice as long? Fortunately, the answer is no. MatLab first checks to see if the matrix (L and U in this situation) are in triangular form[3], and if they are, it does not perform Gaussian elimination and only performs the appropriate substitution routine. This means that it actually only takes 2 $O\left(n^2\right)$ operations for LU decomposition as opposed to $O\left(n^3\right)+O\left(n^2\right)$ for the A\b. Therefore, the total cost for this operation is $O\left(n^3\right)$ once to obtain L and U, and then, $2\,O\left(n^2\right)$ for each solve. This is a significant improvement over A\b.

Elliptic Solve Using GMRES, BICGSTAB

The last method to explore is an iterative technique to solve the elliptic equation. This involves implementing the Jacobi scheme[4] until a tolerance is met.

1. Guess initial condition vector.
2. Iterate Jacobi scheme.
3. Check for convergence.

Note that this scheme it guarantee to converge only if the Matrix is strictly diagonal dominant. Strictly diagonal dominant is defined as

$$\left|a_{kk}\right| > \sum_{j=1,\, j\neq k}^{N} \left|a_{kj}\right|$$

Equation 17

[3] See page 44 of Reference [1].
[4] See page 45 of Reference [1].

This states that the absolute value of the diagonal term must be strictly greater than the sum of the absolute value of the rest of the elements on the same row. This test can be applied to the A matrix which has the same structure on each row, so this only needs to be performed for one row.

$$|a_{kk}| > \sum_{j=1, j\neq k}^{N} |a_{kj}|$$

$$|-4| > |1| + |1| + |1| + |1|$$

$$4 = 4$$

From this one can see that the A matrix is not strictly diagonal dominant, but it is actually very close since the diagonal term is equal to the sum of the other elements in that row. This means that the scheme is not guarantee to converge, but there is a good chance that it will.

Vorticity Coefficient[5]

In this problem, there is viscous diffusion in the advection/diffusion equation, which serves to reduce the vorticity on the computational domain over time. One can define a vorticity coefficient to measure the overall vorticity over the entire computational domain at any time, t_o. The equation for the vorticity coefficient for a continuous domain is shown below.

$$C_\omega = \frac{\int_0^{Y_L}\int_0^{X_L} |\omega(x, y, t_o)| dxdy}{X_L Y_L}$$

Equation 18

For a discrete computational domain, we can develop a discrete form of Equation 18 to obtain the vorticity coefficient for a discrete domain.

[5] The author derived this coefficient independently. This coefficient has not been tested extensively and therefore, should only be used for preliminary estimates and analysis.

$$C_{\omega} = \frac{\sum_{n=0}^{N_L} \sum_{m=0}^{M_L} \left| \omega\left(m\frac{X_L}{M_L}, n\frac{Y_L}{N_L}, t_o \right) \right|}{M_L N_L}$$

Equation 19

Effectively, this coefficient sums the absolute value of the vorticity over the entire computational domain at a certain time. It is then normalized by the number of sample points that were used so that the coefficient of vorticity should be independent of the density of the mesh.

Implementation

Control Flow

This problem can solved using MatLab given that it is a highly repetitious series of operations. The general control flow of the code is shown below in Figure 2.

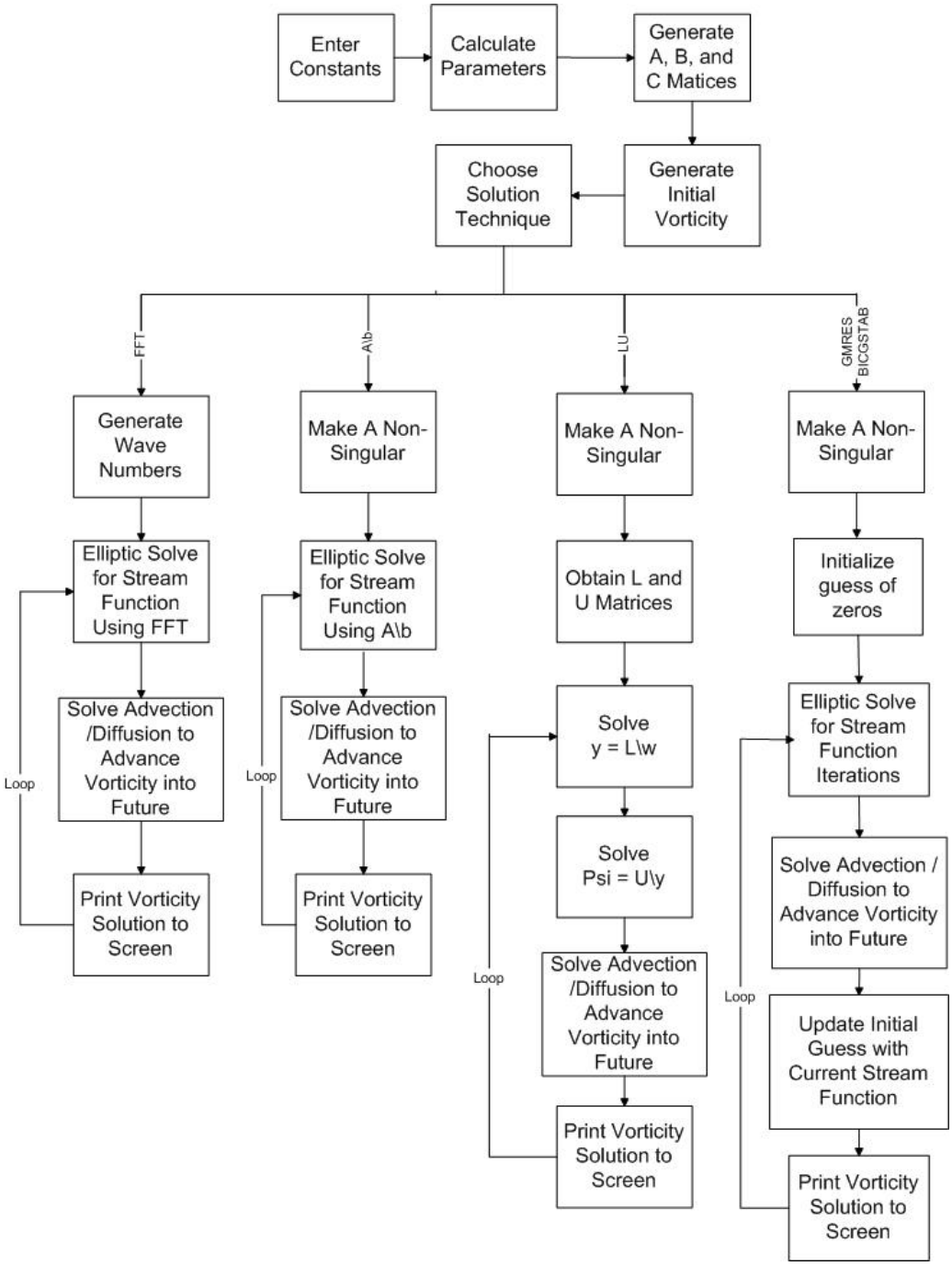

Enter Constants
Calculate Parameters
Generate A, B, and C Matices
Generate Initial Vorticity
Choose Solution Technique
FFT
A\b
LU
GMRES BICGSTAB
Generate Wave Numbers
Elliptic Solve for Stream Function Using FFT
Solve Advection /Diffusion to Advance Vorticity into Future
Print Vorticity Solution to Screen
Loop
Make A Non-Singular
Elliptic Solve for Stream Function Using A\b
Solve Advection /Diffusion to Advance Vorticity into Future
Print Vorticity Solution to Screen
Loop
Make A Non-Singular
Obtain L and U Matrices
Solve y = L\w
Solve Psi = U\y
Solve Advection /Diffusion to Advance Vorticity into Future
Print Vorticity Solution to Screen
Loop
Make A Non-Singular
Initialize guess of zeros
Elliptic Solve for Stream Function Iterations
Solve Advection / Diffusion to Advance Vorticity into Future
Update Initial Guess with Current Stream Function
Print Vorticity Solution to Screen
Loop

Figure 2: Control flow for code

All of the methods here rely on the assumption that the stream function does not change much when compared to the vorticity.[6] Therefore, it becomes possible to solve for the stream function, then use ode23 or ode45 to update the vorticity using a constant stream function. After a small time, say 0.5 seconds, the stream function is updated and the process repeats itself.

As will be shown later, it is actually faster to update the stream function at every time step inside the ode call using the FFT. The code for this is also simpler since the ode call can be made from 0 to the final time all at once instead of coming out of the ode call and updating the stream function. The stream function is then updated inside the ode call. The process inside the ode call is then somewhat along the lines of

1. Reshape vorticity into a matrix
2. Transform to Fourier domain in both directions using FFT ($\tilde{\tilde{\omega}}$)
3. Solve for $\tilde{\tilde{\psi}}$ using Equation 14.
4. Transform back to spatial domain using IFFT
5. Reshape stream function matrix into a column vector.
6. Form differential equation using column form of vorticity, stream function, A, B, and C matrices using Equation 11.

It will be shown later in the Results section that this is actually a faster technique to solve this problem.

[6] See Results section for validation of this assumption.

Computation Difficulties

This is the general control flow of the code. In addition to this, there are several computational difficulties that must be overcome.

The first of these problems involves the elliptic solve for $A\overline{\psi} = \overline{\omega}$. Recall that the A matrix is a differentiation matrix, and therefore, it is singular since it is defined for any arbitrary constants of integration. The singularity of the matrix can be checked using the condition number, which is the ratio of the largest singular value over the smallest singular value from the singular value decomposition. Large condition numbers imply a singular matrix.

```
>> cond(A)

ans =

  4.0993e+016
```

As can be seen, this is most definitely singular. Therefore, the elliptic solve will not be possible unless the A matrix is made non-singular. The most tempting thing to do at this point would be to change the first element of the matrix to 0. However, this will cause a problem if this is done for the LU decomposition. If the first element is zero, then when MatLab performs the decomposition, it will have a 0 in the first pivot spot and it will then have to switch rows. This can be seen if looking at the permutation matrix, P. This is shown below if A(1,1) = 0.

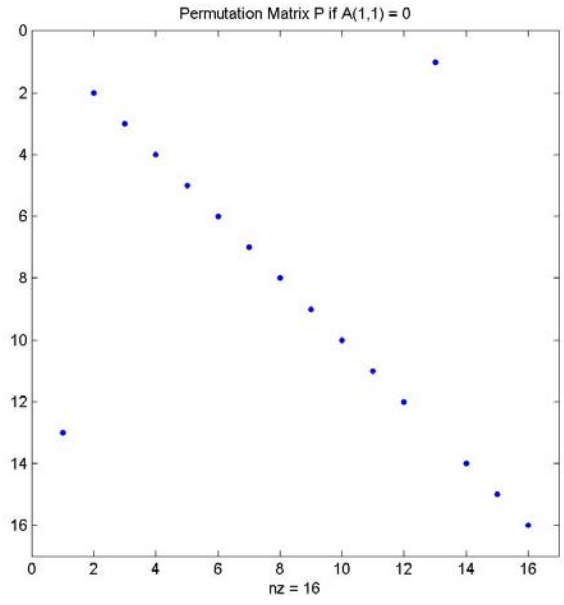


Figure 3: Permutation matrix if A(1,1) = 0

As can be seen, this is not zero, therefore, Equation 15 becomes

$$\bar{y} = P^{-1}L \setminus \bar{\omega}$$

As can be seen, this is not desirable since it requires calculating an inverse of the P matrix and then multiplying it the matrix L. This greatly increases computational time well beyond the requirements for the A\b technique. Therefore, the A matrix must be made non-singular in a fashion where the permutation matrix is the identity matrix. By changing A(1,1) = 1, the P matrix shown in Figure 4 is obtained.

As can be seen, this yields P = I. The matrix is also full rank as shown by the condition number.

```
>> cond(A2)

ans =

  140.6667
```

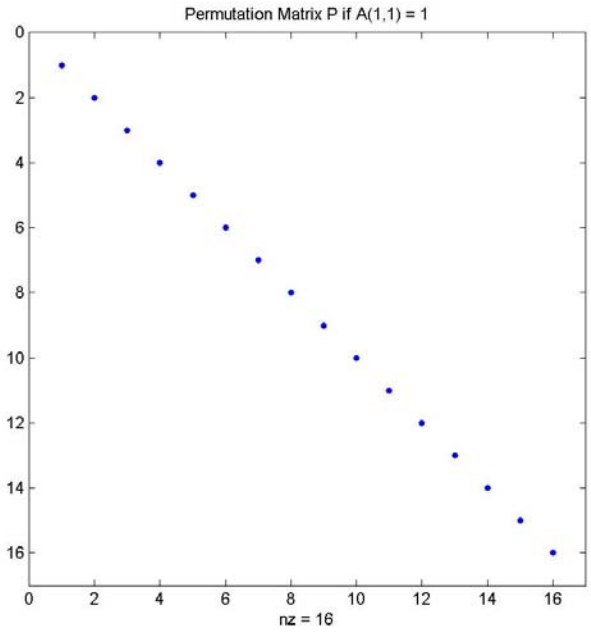


Figure 4: Permutation matrix if A(1,1) = 1

The next problem involves the use of the Fast Fourier Transform. The Fast Fourier transform relies on an infinite sum of cos and sin functions with varying coefficients and wave numbers. However, for a numerical solution, only a finite number of cos and sin function can be used. Furthermore, only a certain wave numbers are eligible which satisfy the periodic boundary conditions. The MatLab function FFT2 can be used to calculate the coefficients of each of the cos and sin terms, but it assumes that the domain is a periodic from -2π to 2π. However, the domain for this problem is most likely not -2π to 2π. Instead, it is from –L/2 to L/2 in both the x and y direction. This means that the wave numbers, which correspond to the new domain, must be calculated. These wave numbers can be calculated using Equation 20.

$$k_i = \frac{2\pi}{L} i \qquad for\ i = -\frac{m}{2}, -\frac{m}{2}+1, ..., -1, 0, 1, ..., \frac{m}{2}-1, \frac{m}{2}$$

Equation 20

The wave numbers calculated from Equation 20 can be used in $\cos(k_n x)$ and $\sin(k_n x)$ and these functions will satisfy the periodic conditions over the domain from –L/2 to L/2. The first three wave numbers after zero and their corresponding functions are shown below in Figure 5 and Figure 6 for a computational domain from –10 to 10.

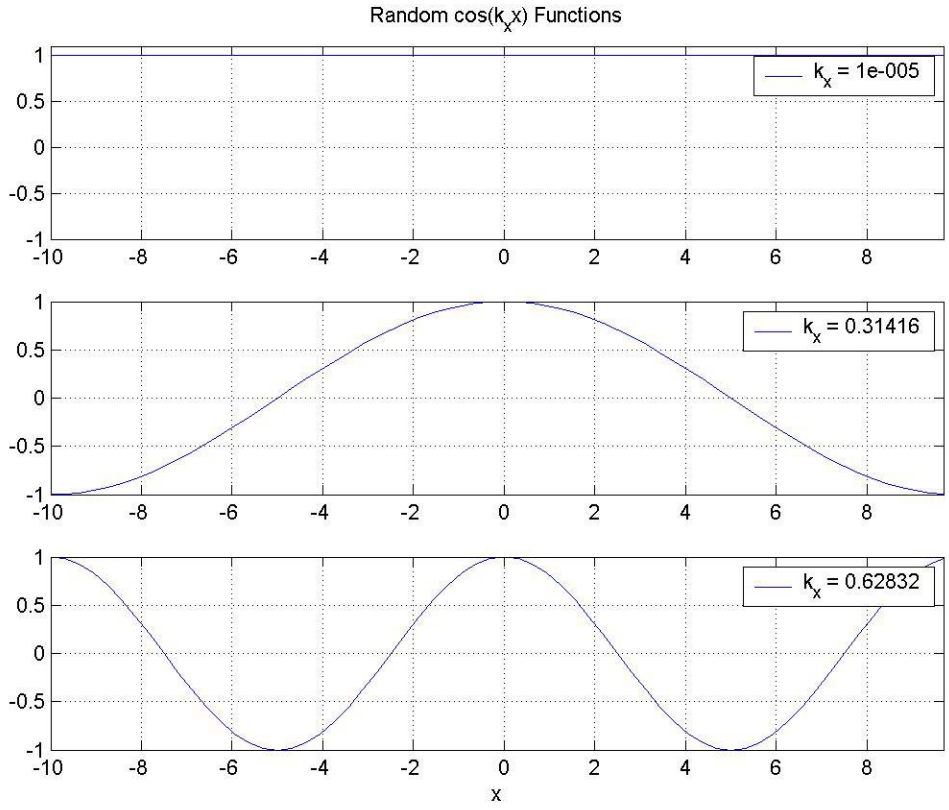


Figure 5: $\cos(k_n x)$ showing that wave numbers satisfy periodic boundary conditions

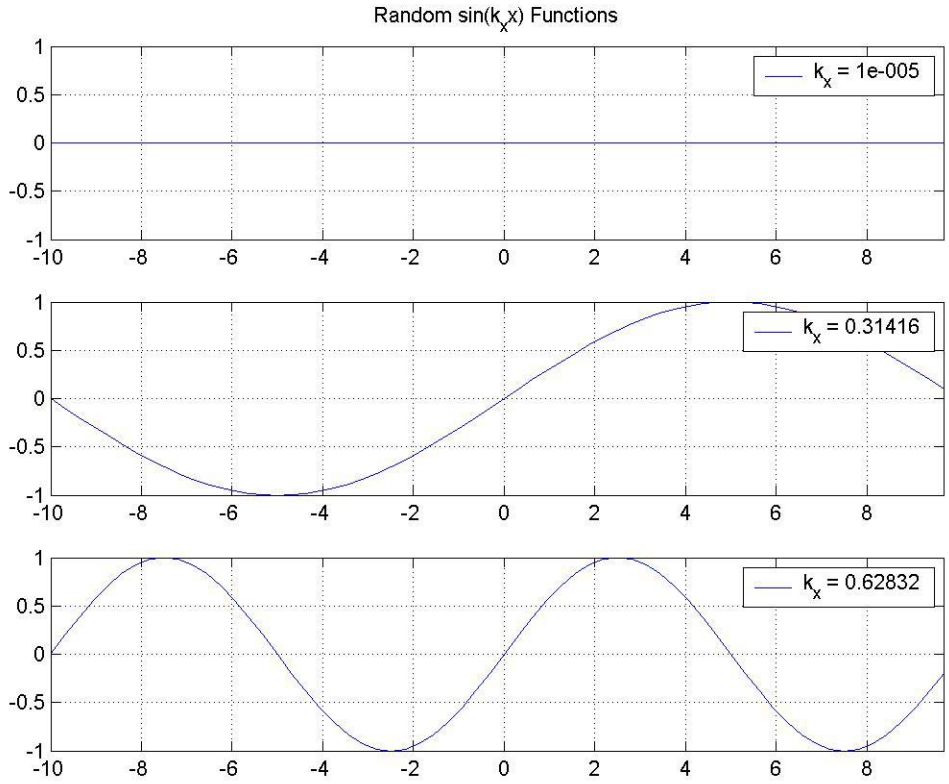


Figure 6: sin(k_nx) showing that wave numbers satisfy periodic boundary conditions

As can be seen, Equation 20 successfully calculates the correct wave numbers for the new domain from –L/2 to L/2 since all the resulting cos and sin functions satisfy the periodic boundary conditions.

The other problem to address with the FFT is the fact that it may be possible to divide by zero. Recall that the stream function transformed in both x and y directions is given by Equation 14 which is repeated here for convenience

$$\tilde{\tilde{\psi}} = \frac{-\tilde{\tilde{\omega}}}{k_x{}^2 + k_y{}^2}$$

From Figure 5 and Figure 6, one can see that $k_x = k_y = 0$ is both an eligible and necessary wave number (since it produces a constant DC term). However, in this case, it can then make it possible for Equation 14 to be undefined since both k_x and k_y can equal zero at the same time. Therefore, the easiest solution to this is instead of using the first wave number as $k_x = k_y = 0$, change it to $k_x = k_y = 10e\text{-}6$. In fact, this was what was done to generate Figure 5 and Figure 6 with very satisfactory results.

The last problem to address with the FFT is that MatLab expects the wave numbers to be shifted about the midpoint. In other words, it expects the first element of the k_x and k_y vectors to be zero and the last element to be near zero as well. In order to perform this shift, we use

```
k = 2*pi/L*[0:(m/2-1) (-m/2):-1]
```

This produces a array, which has wave numbers ordered in the correct fashion. The wave number vs. array position is shown below in Figure 7.

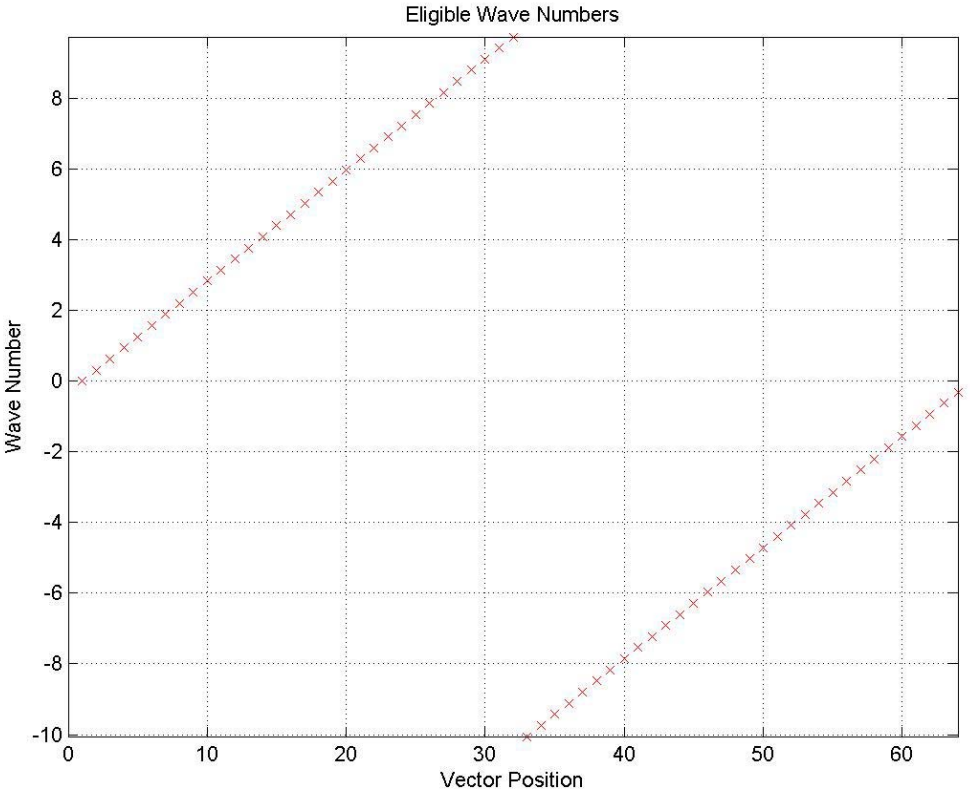


Figure 7: Wave numbers vs. vector position

Results

The vorticity as a function of time can now be simulated using any of the above-mentioned methods. The only portion of the simulation that can vary is the choice of elliptic solver. The six different types are now discussed. Since it is the most accurate and efficient, the results form the Fast Fourier Transform will be discussed first.

Fast Fourier Transform

Recall that all of the above mentioned methods rely on the assumption that the stream function is roughly constant over the small time interval that the vorticity is solved for. Recall from Figure 2 that the initial stream function is solved for and then it is held constant over the time interval where ode23 solves the advection-diffusion equation. At the end of this time period, the stream function is updated and the process repeats itself. The assumption that the stream function changes slowly when compared to the vorticity needs to be verified. By using a computational domain from –10 to 10 (L = 20) in both the x and y direction and using 128 modes in the x and y direction, the simulation can be performed using an initial condition of a stretched Gaussian of the form $\omega(x,y,0) = \exp\left(-2x^2 - y^2/L\right)$. Plots of both the vorticity and the corresponding stream function are shown in Figure 8 and Figure 9, respectively.

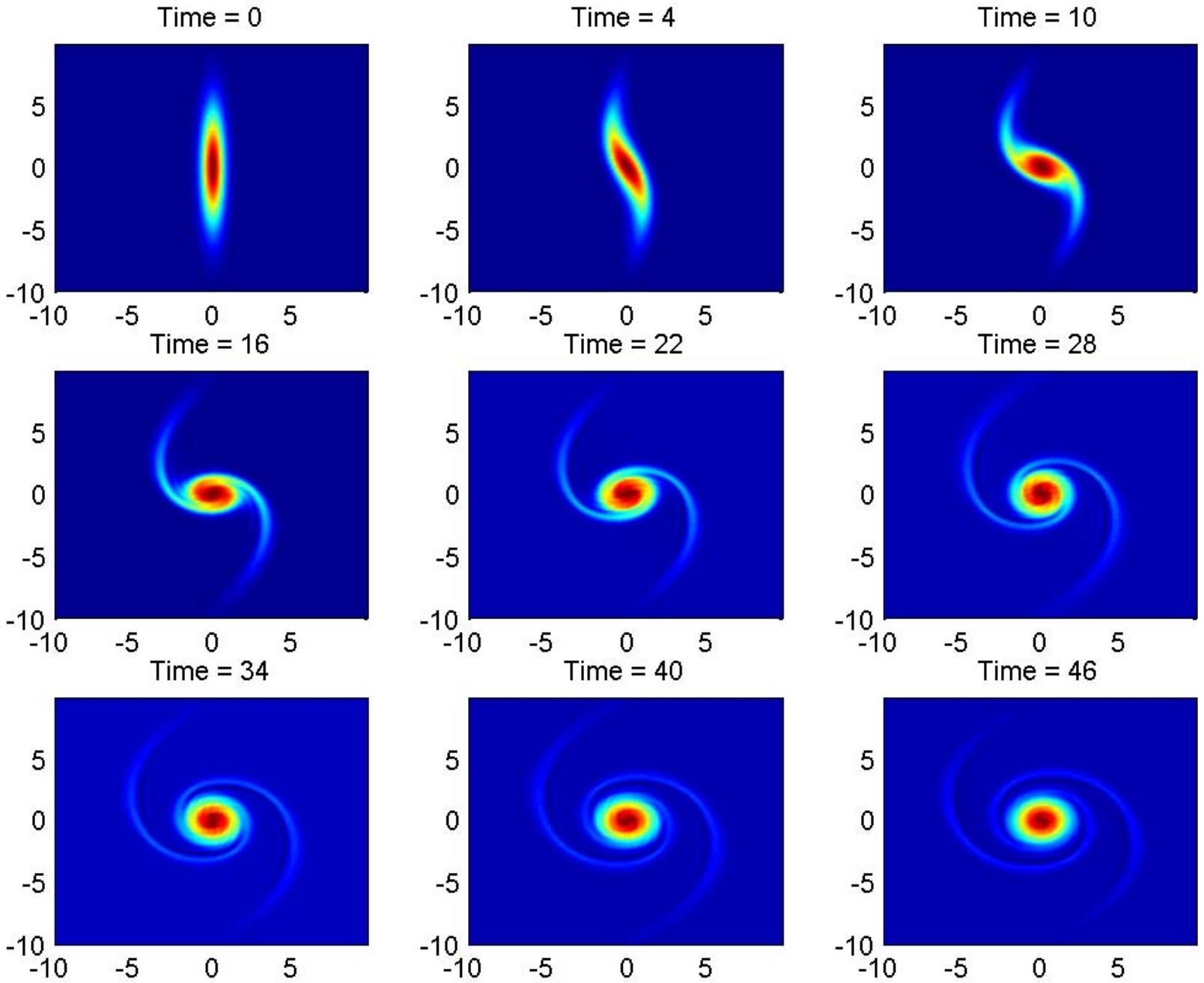


Figure 8: Vorticity as a function of time with stretched Gaussian as initial condition

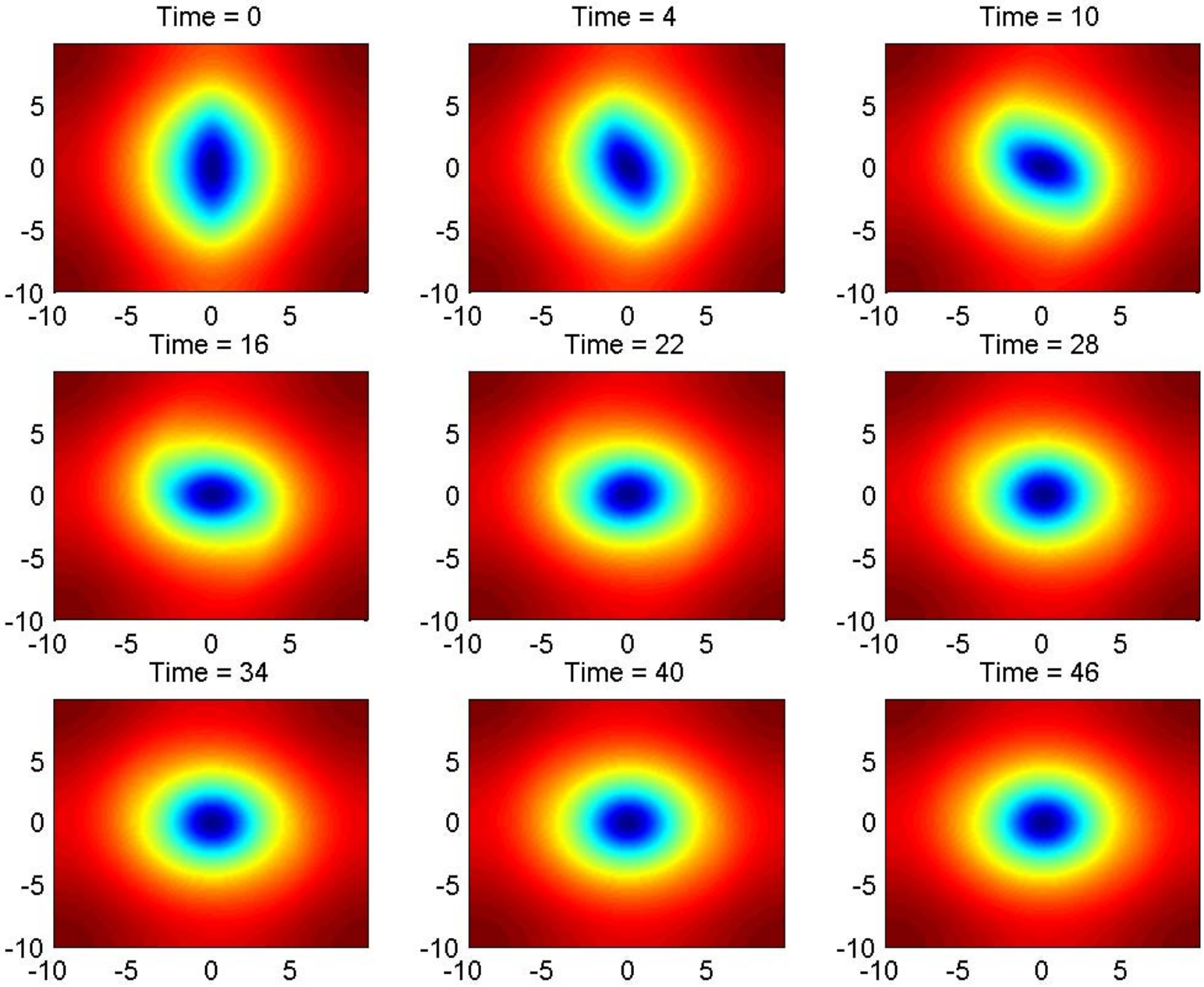


Figure 9: Stream function as a function of time with stretched Gaussian as initial condition

As can be seen, the vorticity evolves much faster than the stream function. In fact, it appears at later times that the stream function does not change at all. Therefore, it is safe to assume that the stream function is constant over a small change in time when compared to the vorticity.

In order to measure the speed of this method, we can perform the same operation with the same initial conditions and parameters over a much smaller time interval of $t \in [0,1.5]$ and update the stream function every 0.5 seconds. An excerpt from the screen diagnostics is shown below.

```
Now solving using FFT...

Updating only at discrete steps.

Solving for t = 0 sec to t = 0.5
    Time for elliptic solve w/ FFT = 0.047 sec.
    Time to solve ode call = 2.891 sec.
    11 time steps used for ode call.

    Section time = 3.078 sec.

Solving for t = 0.5 sec to t = 1
    Time for elliptic solve w/ FFT = 0.016 sec.
    Time to solve ode call = 2.859 sec.
    11 time steps used for ode call.

    Section time = 2.984 sec.

Solving for t = 1 sec to t = 1.5
    Time for elliptic solve w/ FFT = 0.016 sec.
    Time to solve ode call = 2.875 sec.
    11 time steps used for ode call.

    Section time = 2.985 sec.

TOTAL TIME = 9.609 sec.
```

As can be seen, this is a very fast solver as it takes a very small amount of time to perform the elliptic solve. If this were run over a longer time period, it would be seen that it takes an average of approximately 0.031 seconds to perform the elliptic solve.

A\b

The same time interval and conditions can be solved for using the A\b method. An excerpt from the screen diagnostics are shown below

```
Now solving using A\b...

Solving for t = 0 sec to t = 0.5
    Time for elliptic solve with A\b = 1.875 sec.
    Time to solve ode call = 1.656 sec.
    11 time steps used for ode call.

    Section time = 3.531 sec.

Solving for t = 0.5 sec to t = 1
    Time for elliptic solve with A\b = 1.484 sec.
    Time to solve ode call = 1.719 sec.
    11 time steps used for ode call.

    Section time = 3.203 sec.

Solving for t = 1 sec to t = 1.5
    Time for elliptic solve with A\b = 1.484 sec.
    Time to solve ode call = 1.703 sec.
    11 time steps used for ode call.

    Section time = 3.203 sec.

TOTAL TIME = 10.343 sec.
```

As can be seen, this technique is significantly slower than the FFT technique. The average time required for the elliptic solve for this solution is roughly 1.47 seconds for each solution. Notice that one cannot really compare the total time required for each method since this takes into account overhead operational costs like printing to the screen and updating figures.

LU Decomposition

An excerpt from the screen diagnostics for this method is shown below

```
Now solving using LU Decomposition...

TIME TO OBTAIN L AND U = 10.671 sec.

Solving for t = 0 sec to t = 0.5
    Time for forward substitution = 0.281 sec.
    Time for backward substitution = 0.407 sec.
    Time for elliptic solve with LU = 0.859 sec.
    Time to solve ode call = 1.906 sec.
    11 time steps used for ode call.

    Section time = 2.828 sec.

Solving for t = 0.5 sec to t = 1
    Time for forward substitution = 0.281 sec.
    Time for backward substitution = 0.297 sec.
    Time for elliptic solve with LU = 0.609 sec.
    Time to solve ode call = 1.704 sec.
    11 time steps used for ode call.

    Section time = 2.313 sec.

Solving for t = 1 sec to t = 1.5
    Time for forward substitution = 0.281 sec.
    Time for backward substitution = 0.297 sec.
    Time for elliptic solve with LU = 0.593 sec.
    Time to solve ode call = 1.703 sec.
    11 time steps used for ode call.

    Section time = 2.312 sec.

TOTAL TIME = 18.562 sec.
```

As can be seen, it is a fairly costly operation to break the A matrix into the L and U matrix. However, this is only required to be performed once, and therefore, if the solution is solved over a long time period, this cost becomes negligible. This technique also appears to stabilize as the iterations proceed. In fact, if the operations were carried out over more intervals, it would be seen that the technique settles out around $t_{required}$ = 0.593 seconds.

BICGSTAB

The BICGSTAB (BiConjugate Gradients Stabilized) method is an iterative technique used to perform the elliptic solve. With this iterative method, it is important to define a tolerance for the convergence. Given that the discretization of the A, B, and C matrices are only accurate of $O\left(\delta^2\right)$, the tolerance for the iteration for the elliptic solve should be of the same order. In this case, we choose the tolerance to be $0.75\delta^2 \approx 0.018$, which is slightly more accurate than the finite difference discretization to ensure that no additional errors are being introduced with the elliptic solve. Furthermore, the default number of maximum iterations must be changed from the default and increased to 500.

An excerpt from the screen diagnostics for this method is shown below

```
Now solving using BICGSTAB...

Solving for t = 0 sec to t = 0.5
    439 iterations required to converge to 0.018311
    Time for elliptic solve with BICGSTAB = 18.609 sec.
    Time to solve ode call = 1.719 sec.
    11 time steps used for ode call.

    Section time = 20.343 sec.

Solving for t = 0.5 sec to t = 1
    22.5 iterations required to converge to 0.018311
    Time for elliptic solve with BICGSTAB = 0.953 sec.
    Time to solve ode call = 1.719 sec.
    11 time steps used for ode call.

    Section time = 2.687 sec.

Solving for t = 1 sec to t = 1.5
    29.5 iterations required to converge to 0.018311
    Time for elliptic solve with BICGSTAB = 1.266 sec.
    Time to solve ode call = 1.718 sec.
    11 time steps used for ode call.

    Section time = 3 sec.

TOTAL TIME = 26.718 sec.
```

One interesting observation to make is that the first elliptic solve requires the most iterations and most time to perform. This makes sense considering what that for the first elliptic solve, the initial guess for the stream function is all zeros and then for the subsequent iterations, we use the old stream function as the initial guess. And since the stream function does not change much, all of the iterations that follow after the first one

will be much more efficient. To illustrate this, the relative residual vs. the iteration number for the first three iterations are shown below in Figure 10 through Figure 12.

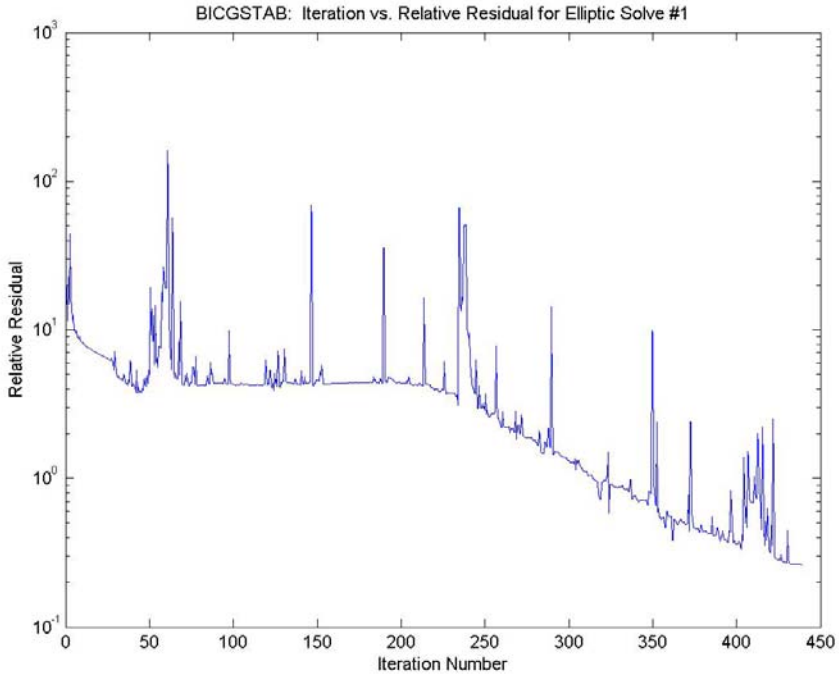


Figure 10: Relative residual vs. iteration number for 1st elliptic solve using BICGSTAB

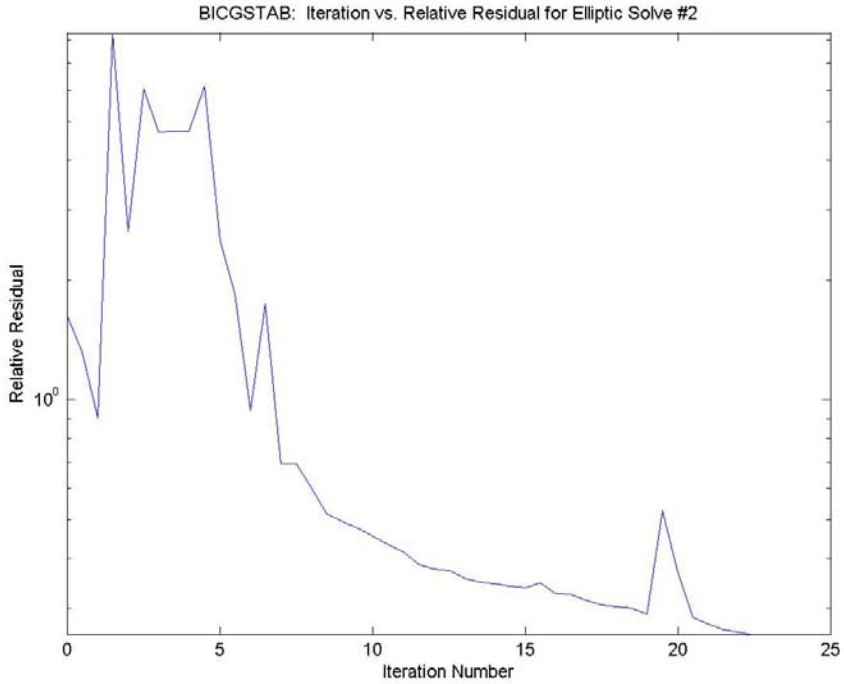


Figure 11: Relative residual vs. iteration number for 2nd elliptic solve using BICGSTAB

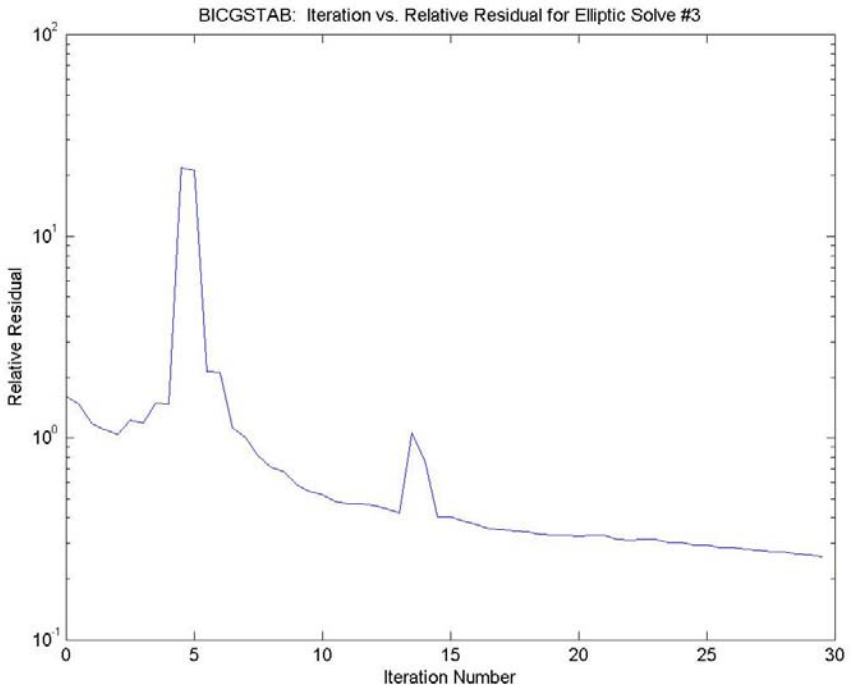


Figure 12: Relative residual vs. iteration number for 3rd elliptic solve using BICGSTAB

As can be seen, the first iteration takes the most iterations to converge since it must start from an initial guess of all zeros. All of the subsequent elliptic solves are much faster since the initial guess is the stream function at the previous time. One possible method to improve this is to start the initial guess from something other than zero. However, since this only improves the time of the first iteration, this would not be much of an improvement for the overall total time required.

The average time required for the elliptic solve using this method was $t_{required}$ = 1.123 seconds.

GMRES

Another iterative technique is the GMRES technique (Generalized Minimum Residual). It functions much like BICGSTAB. An excerpt from the screen diagnostics for this method is shown below

```
Now solving using GMRES...

Solving for t = 0 sec to t = 0.5
    l  216 iterations required to converge to 0.018311
    Time for elliptic solve with GMRES = 62.56 sec.
    Time to solve ode call = 4.5 sec.
    11 time steps used for ode call.

    Section time = 67.185 sec.

Solving for t = 0.5 sec to t = 1
    l  23 iterations required to converge to 0.018311
    Time for elliptic solve with GMRES = 1.75 sec.
    Time to solve ode call = 1.718 sec.
    11 time steps used for ode call.

    Section time = 3.484 sec.

Solving for t = 1 sec to t = 1.5
    l  30 iterations required to converge to 0.018311
    Time for elliptic solve with GMRES = 2.062 sec.
    Time to solve ode call = 1.735 sec.
    11 time steps used for ode call.

    Section time = 3.812 sec.

TOTAL TIME = 87.216 sec.
```

As can be seen, a similar phenomenon occurs in this situation. The first iteration takes the longest since it starts with an initial guess of all zeros. The plots of the relative residual vs. iteration number are shown below in Figure 13 through Figure 15.

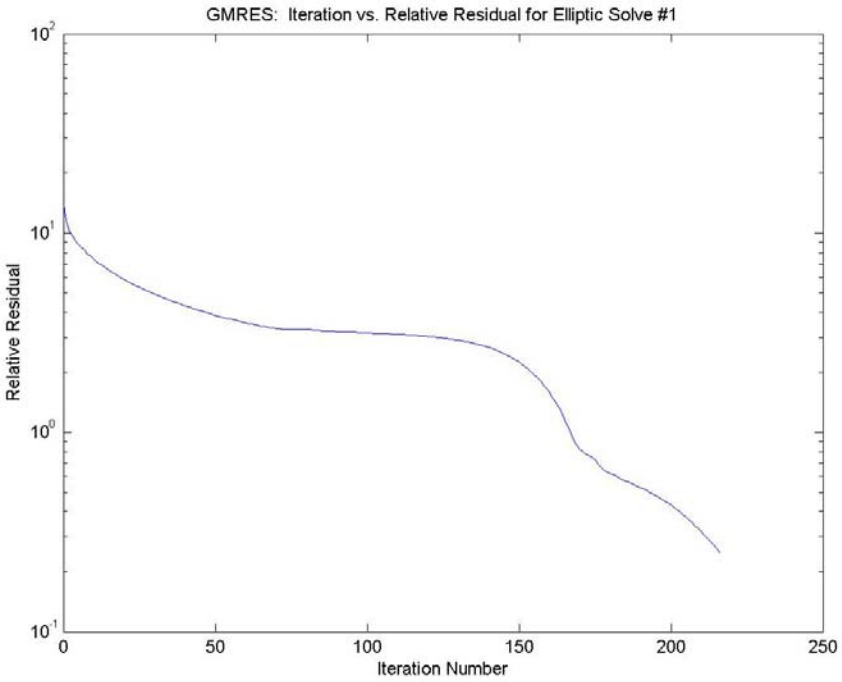


Figure 13: Relative residual vs. iteration number for 1st elliptic solve using GMRES

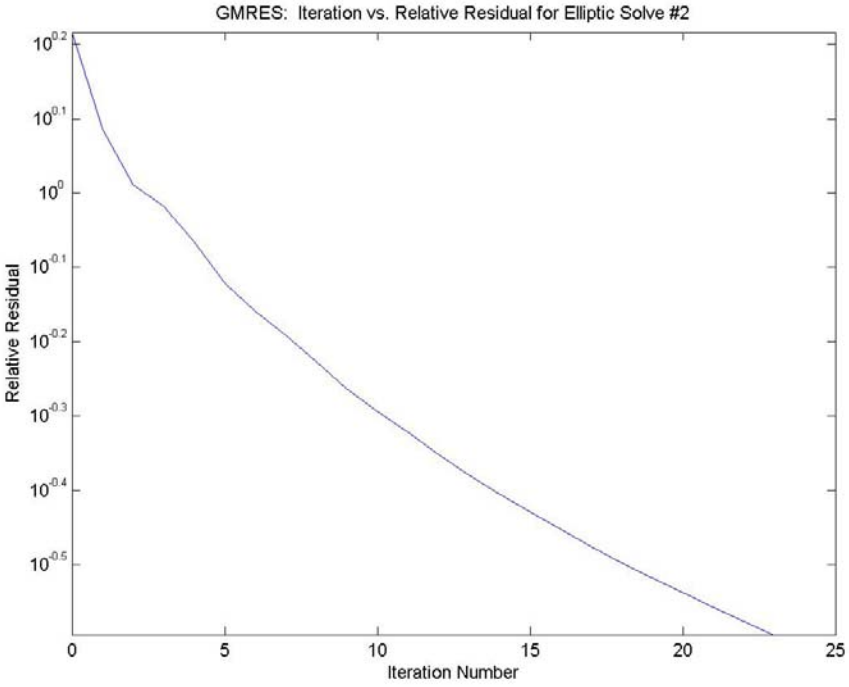


Figure 14: Relative residual vs. iteration number for 2nd elliptic solve using GMRES

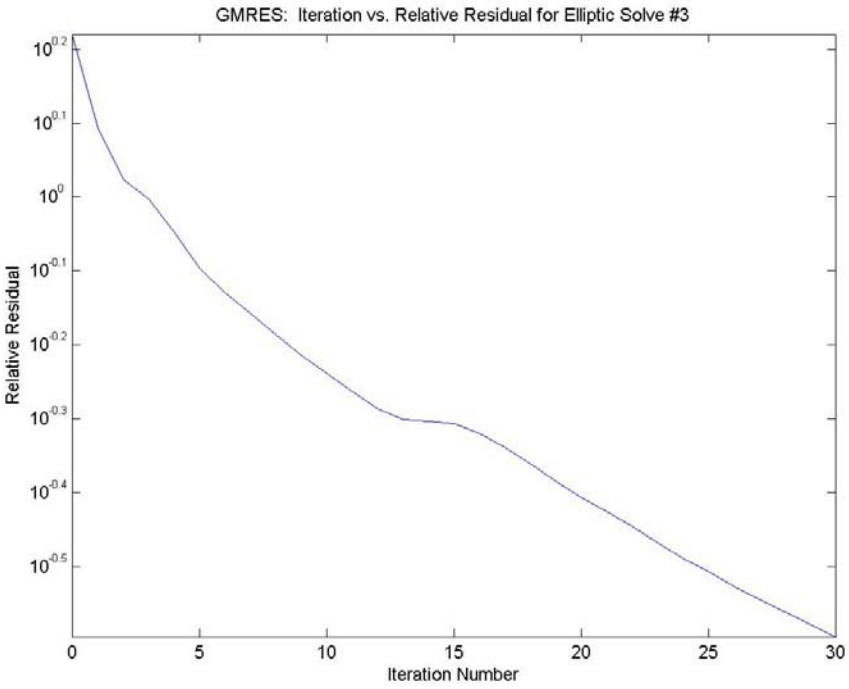


Figure 15: Relative residual vs. iteration number for 3rd elliptic solve using GMRES

As can be seen, this is the same type of phenomenon that occurred with BICGSTAB and can be explained in a similar fashion.

The average time required for the elliptic solve using this method was $t_{required}$ = 2.016 seconds.

The results for the time required for the elliptic solve for the 5 different methods are tabulated below.

Obviously, the FFT is by far that fastest solver. It is nearly 20 times faster than the next fastest solver (LU decomposition). This can be attributed to the fact that the operational cost is $O\left(n\ln\left(n\right)\right)$ opposed to $O\left(n^2\right)$.

Table 2: Average time for elliptic solution using different method

Solution Method	**Average Time for Elliptic Solve (sec)**
FFT	0.031
A\b	1.470
LU Decomposition	0.593
BICGSTAB	1.123
GMRES	2.016

An interesting observation to make is that although it takes only 0.031 seconds to complete the elliptic solve, it takes approximately 3 seconds to solve the advection diffusion equation to advance the vorticity forward by 0.5 seconds. One can see that the ode call requires nearly 100 times more computational time than the elliptic solve. Therefore, in order to decrease computational time, the question becomes how can the time required for the ode call be reduced? This question is addressed in the following section.

Spectral Implementation

Notice, from the screen diagnostics for the FFT, that MatLab uses 11 time steps to solve over the time interval of only 0.5 seconds. Therefore, to solve over 1.5 seconds updating the stream function every 0.5 seconds, then 33 times steps must be used total for the 1.5 time span. If the elliptic solve is placed inside the ode call and the stream function is updated at every time step, then the ode call can simply be made to solve from 0 to the final time all at once (see Implementation section for more information). An excerpt from the screen printout is shown below

```
Solution complete!

    12 time steps used for ode call
    Time to solve ode call = 3.625 sec.

TOTAL TIME = 4.14 sec.

----------------------------------
--PRESS ANY KEY TO PLAY SOLUTION--
----------------------------------
```

As can be seen, this process is much faster than the previous technique even though the elliptic equation was solved at every time step. The explanation for this can be seen when looking at the number of steps used to perform the ode call. Using the technique where the stream function is updated only every 0.5 seconds required 11 steps for each 0.5 second interval. Therefore, the ode call was solved for a total of 33 times. However, in the technique where the stream function is updated every time step and the ode call is performed from 0 to the final time all at once, only 12 total time steps were required. One can see that it is in fact the advection/diffusion solution that seems to takes the most time. This takes roughly 3 seconds whereas the elliptic solve only takes 0.031 seconds. This is why the second solution is so much faster and it will be the method used for simulation of all of the following results.

The first of the simulations to run is a pair of oppositely charged Gaussian vortices that are next to each other. The initial condition is shown below in Figure 16.

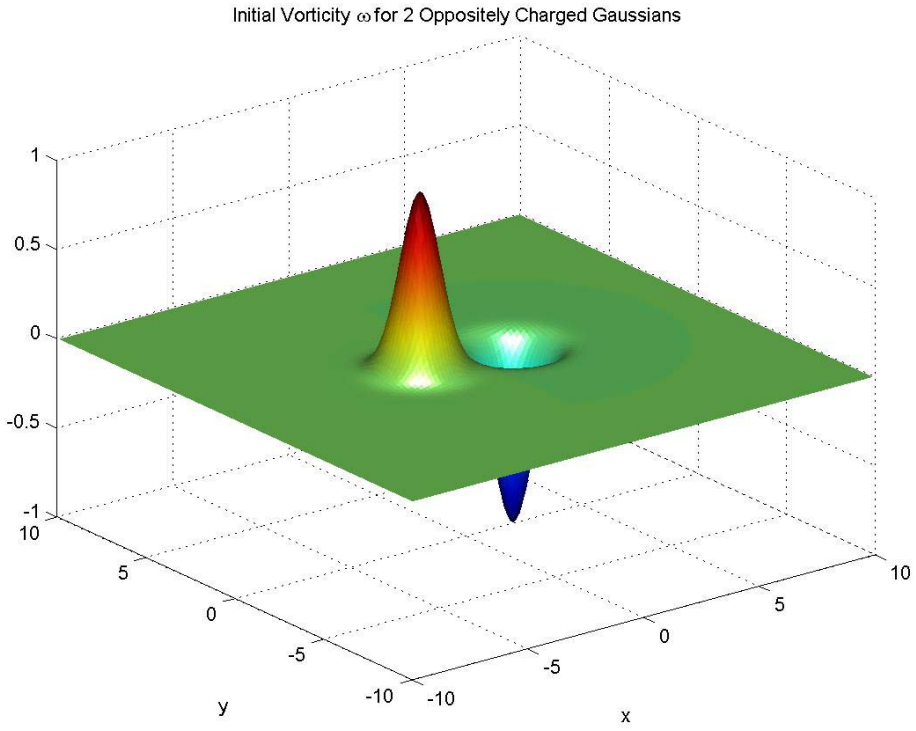


Figure 16: Initial vorticity for two oppositely charged Gaussians

The progression of this initial condition as time progresses is shown below in Figure 17 and Figure 18.

As can be seen, two oppositely charged Gaussians next to each other simply move together. Notice that the periodic boundary conditions can be seen in this example. As the vortices approach the edge of the computational domain, their presence is detected by points on the far side of the computational domain due to the periodic boundary conditions.

The other interesting phenomenon that can be observed here is the effect of the viscous diffusion that is present in this problem. Notice that at t = 0 (frame 1 of Figure 18), the two Gaussians have amplitudes of 1 and -1, respectively. However, after roughly 200 seconds (frame 9 of Figure 18), when the two vortices pass through the same space as when t = 0, the amplitude is much less than 1 and -1. This shows how diffusion serves to reduce the vorticity in the computational domain. Equation 19 can be used to calculate the vorticity coefficient as a function of time from $t \in [0, 250]$ with a viscous diffusion constant of $\upsilon = 0.001$. The first thing to verify is that Equation 19 gives a coefficient that is not dependent on the density of the mesh (ie not dependent on the sample density of x and y sample points, M_L or N_L). A plot of the vorticity coefficient vs. time for different sample densities is shown below in Figure 19.

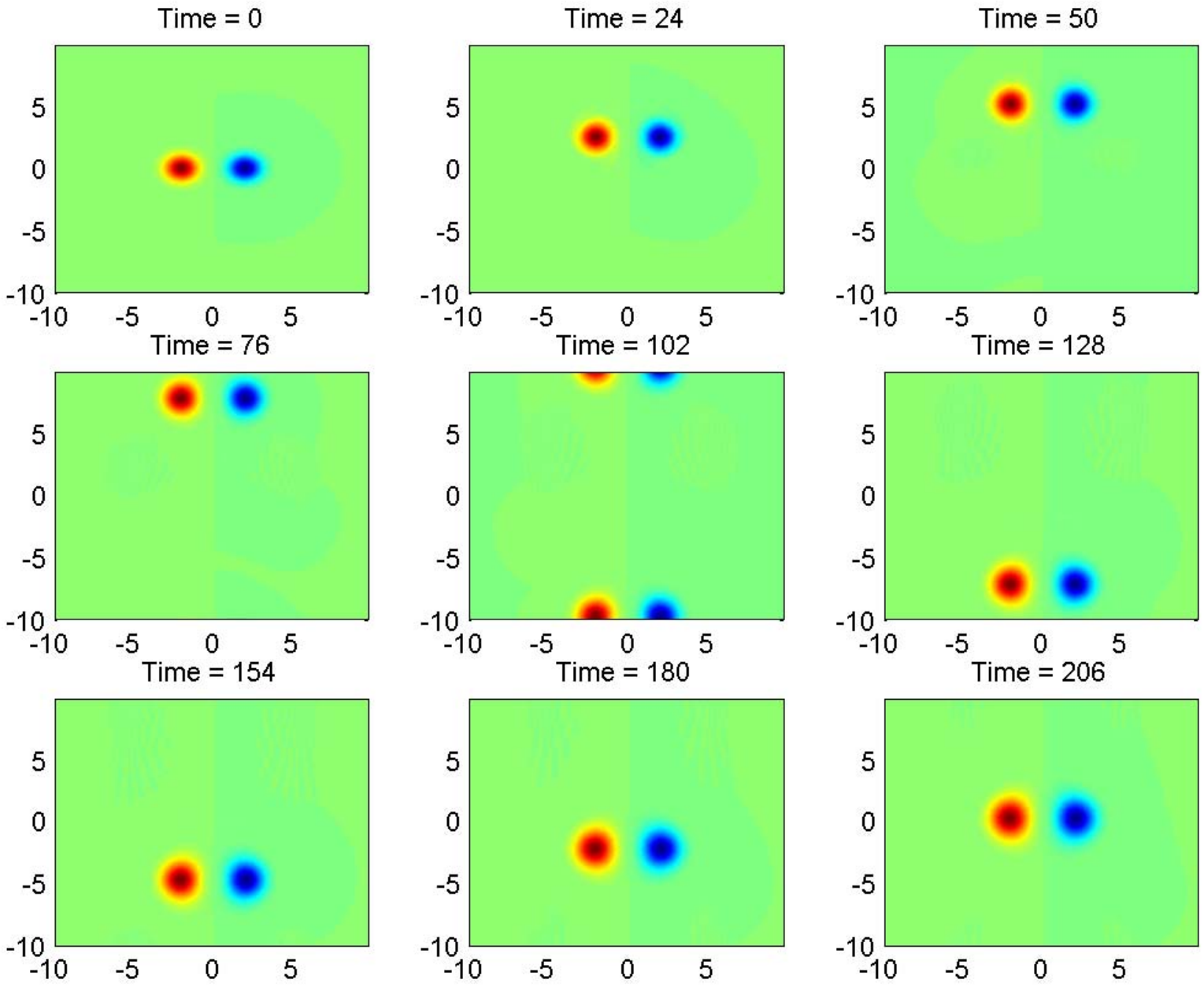


Figure 17: Two opposite Gaussians next to each other (using pcolor function)

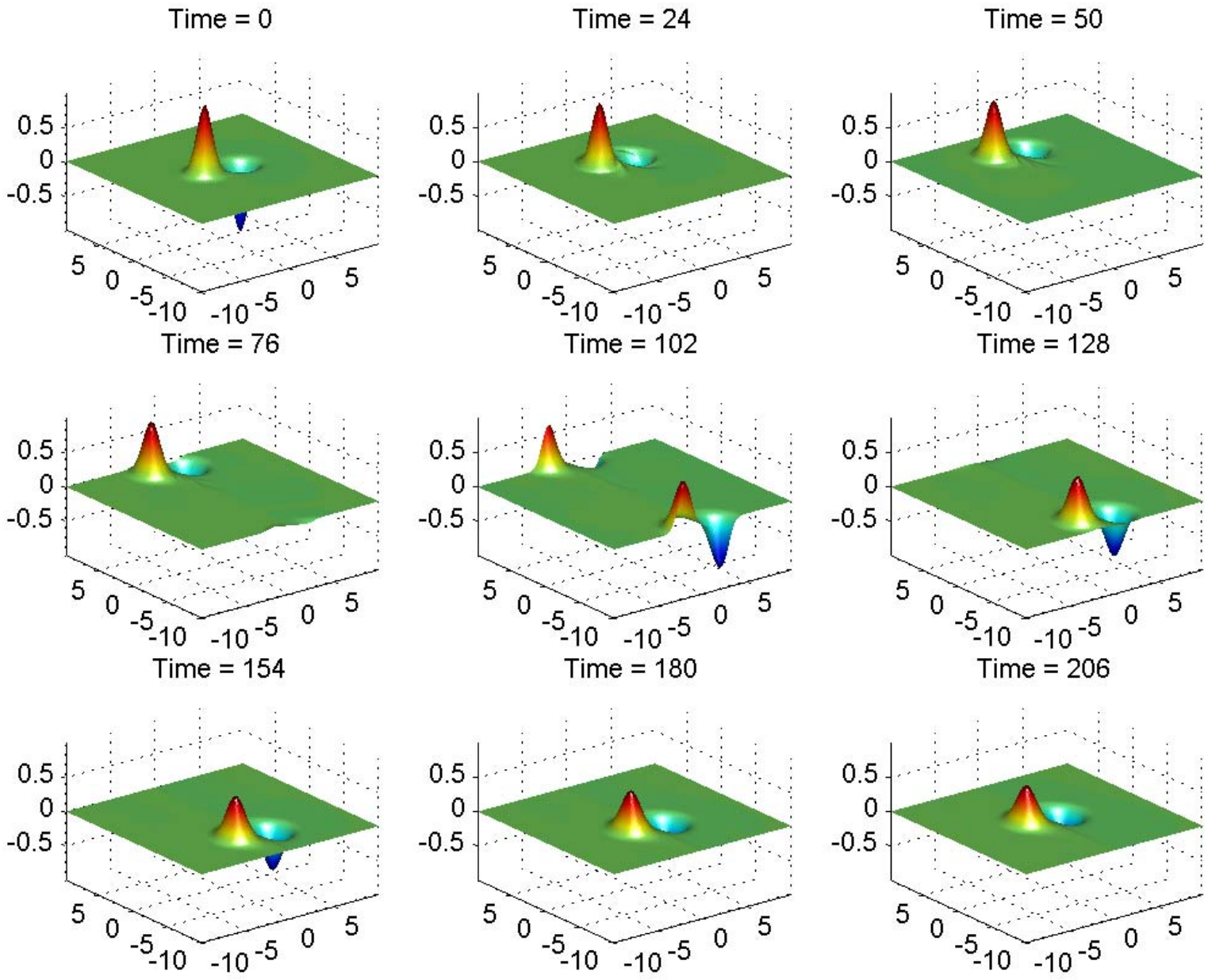


Figure 18: Two opposite Gaussians next to each other (using surf function)

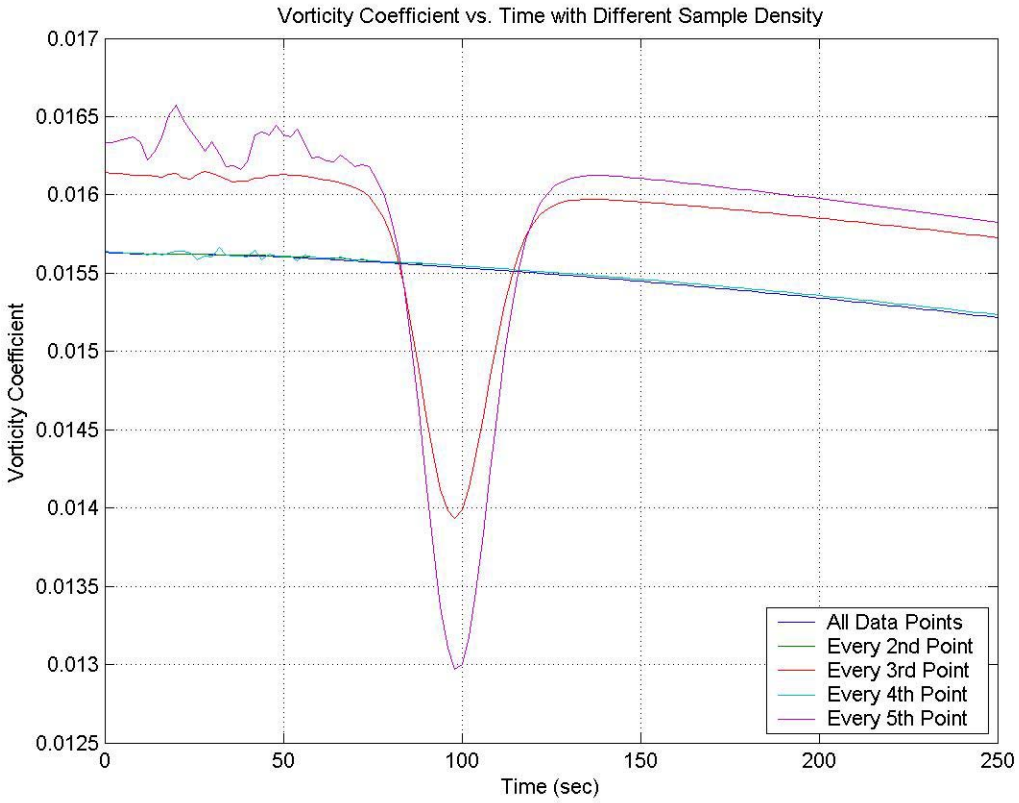


Figure 19: Vorticity coefficient vs. time for different sample densities

As can be seen, it appears that the general trend of a decreasing vorticity coefficient vs. time is captured by all of the cases, except it appears that if a sample density of points where only every 3rd point and every 5th point is used to calculate the coefficient creates an odd behavior at t = 100 seconds. This can be explained by looking at which points are being used to calculate the vorticity coefficient. Most of the vorticity is contained in the two Gaussians. A zoomed in view of the vorticity and the points that are being used to calculate the vorticity coefficient at t = 62 seconds is shown below in Figure 20.

The red x's show points where the vorticity is being contributed to the calculation of the vorticity coefficient. As can be seen, it appears that most of the high intensity vorticity is being used to calculate the vorticity coefficient. Therefore, these contributions make the vorticity coefficient large. The same plot for t = 100 seconds is shown below in Figure 21.

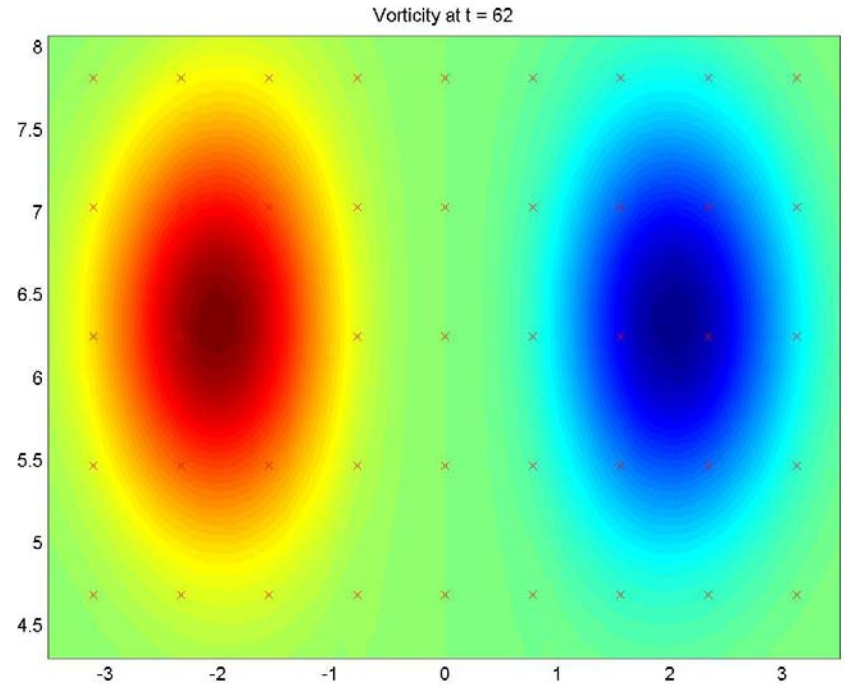


Figure 20: Points (every 5th point) used to calculate vorticity coefficient at t = 62.

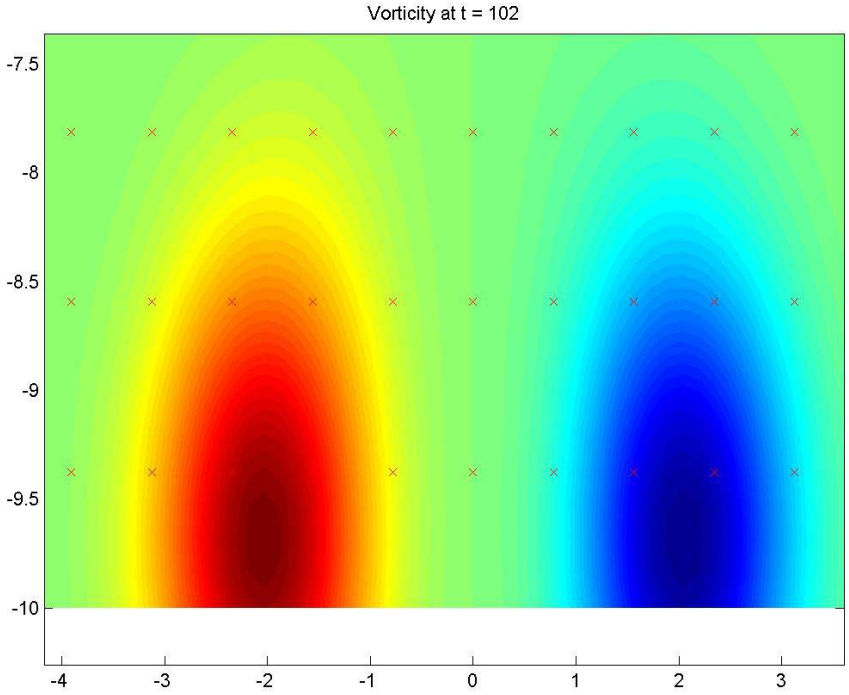


Figure 21: Points (every 5th point) used to calculate vorticity coefficient at t = 102 (lower portion of computational domain)

At can be seen, there is a large drop in the vorticity coefficient because at t = 102, the vortex is half way between transition from one side of the computational domain to the other. Because only every 5th point is being used, the points near the edges are of computational domain are not being used to calculate the vorticity coefficient. This creates a problem because it is in these regions where the vorticity is the largest. This explains the sharp drop in vorticity coefficient at t = 100 seconds. As the vortex moves back into the computational domain, the regions of high vorticity pass under the red x's, and therefore, these regions of high vorticity once again are used to calculate the vorticity coefficient. This explains why the vorticity coefficient increases sharply after t = 125 seconds. This shows that the vorticity coefficient equation (Equation 19) does capture the overall vorticity over the computational domain and it is relatively unaffected by the density of the sample points. However, it is best to use every point in the computational domain to calculate the vorticity coefficient since it will yield the least about of error.

The vorticity coefficient as a function of time with same initial vorticity over the same time interval is shown below for different viscous diffusion coefficients.

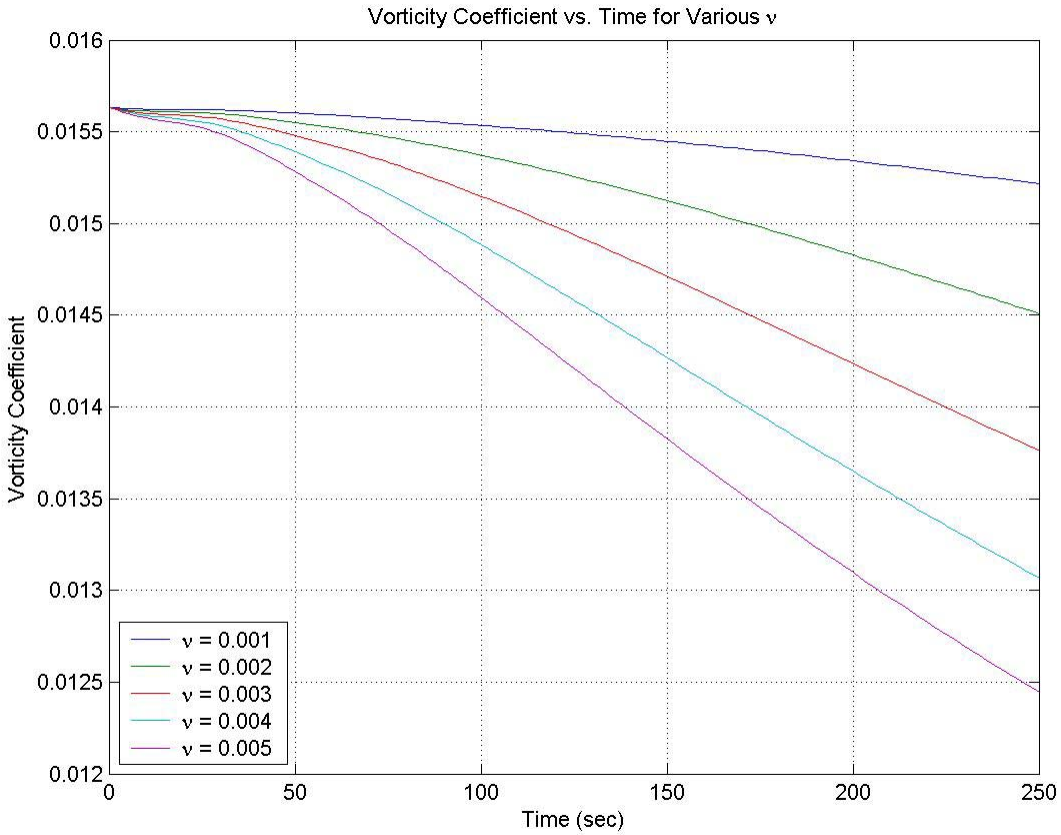


Figure 22: Vorticity coefficient vs. time for various viscous diffusion coefficients

As can be seen, this is a very reasonable result. Regardless of the value of υ, the initial vorticity is the same and as the υ increases, the vorticity dissipates much faster. However, it can be seen that this effect is very small. At steady state, it appears that the vorticity dissipates at a linear rate. Given this linear rate, it appears that it would require 1200 seconds for the vorticity to decay to zero with $\upsilon = 0.005$.

The effect of two vortices that stand next to each other can be evaluated. A plot of the initial condition of this situation is shown below in Figure 23.

The progression of this initial condition as a function of time is shown below in Figure 24 and Figure 25.

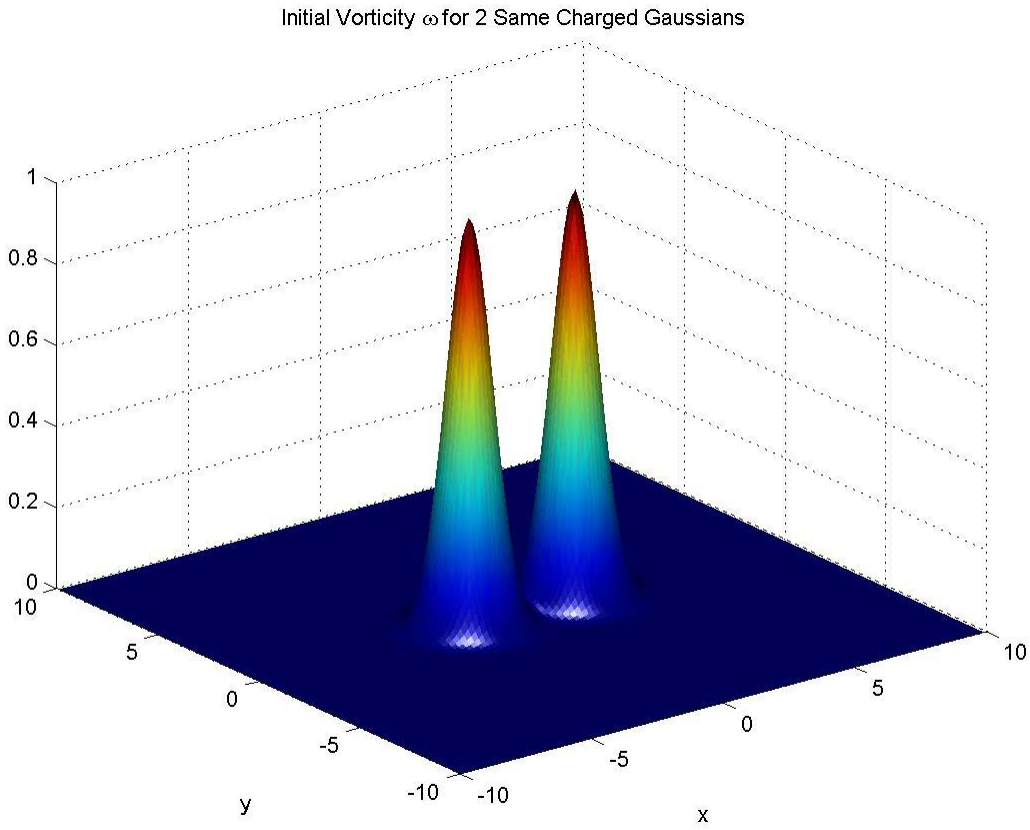


Figure 23: Initial vorticity for two same charged Gaussians

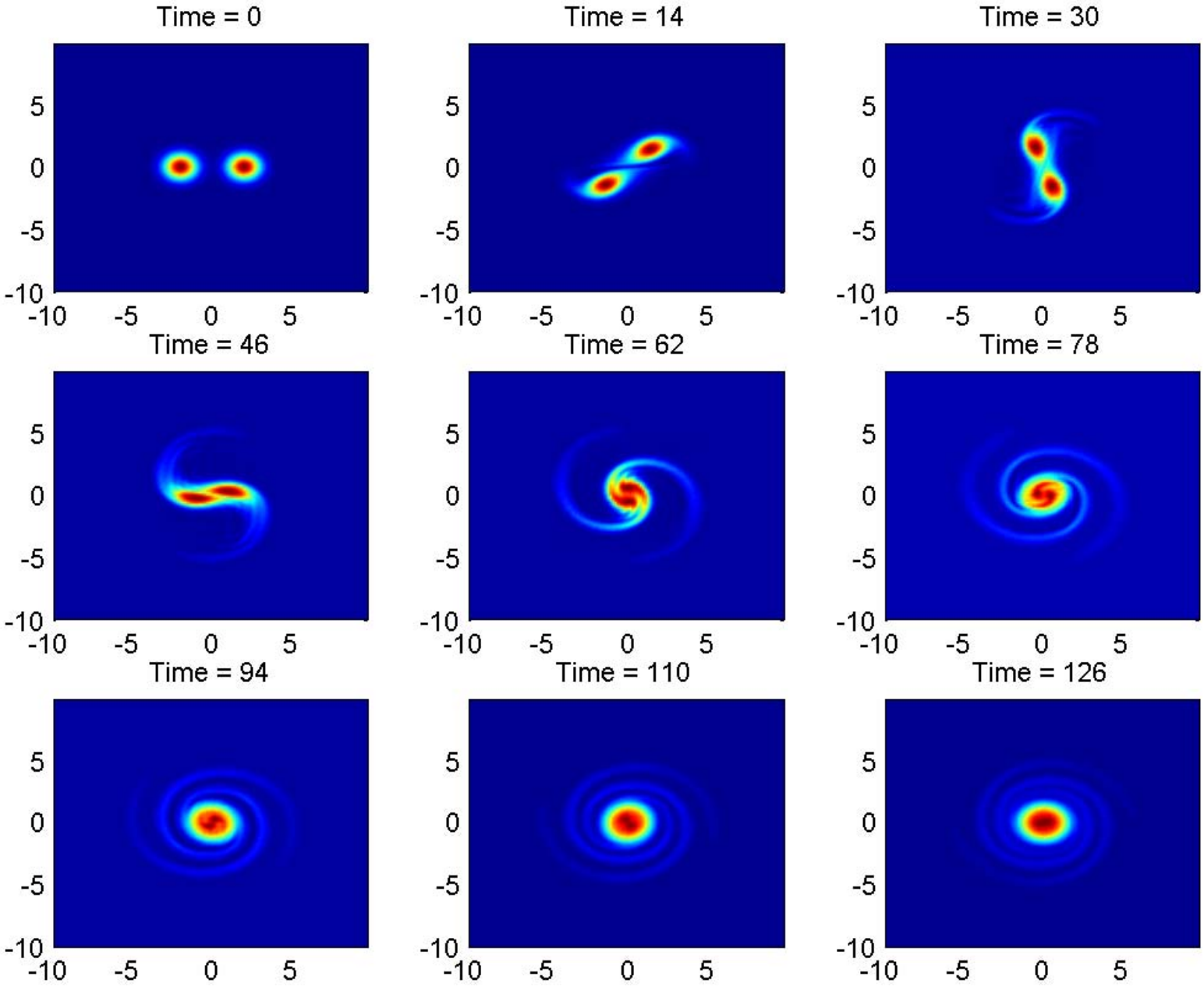


Figure 24: Two same Gaussians next to each other (using pcolor function)

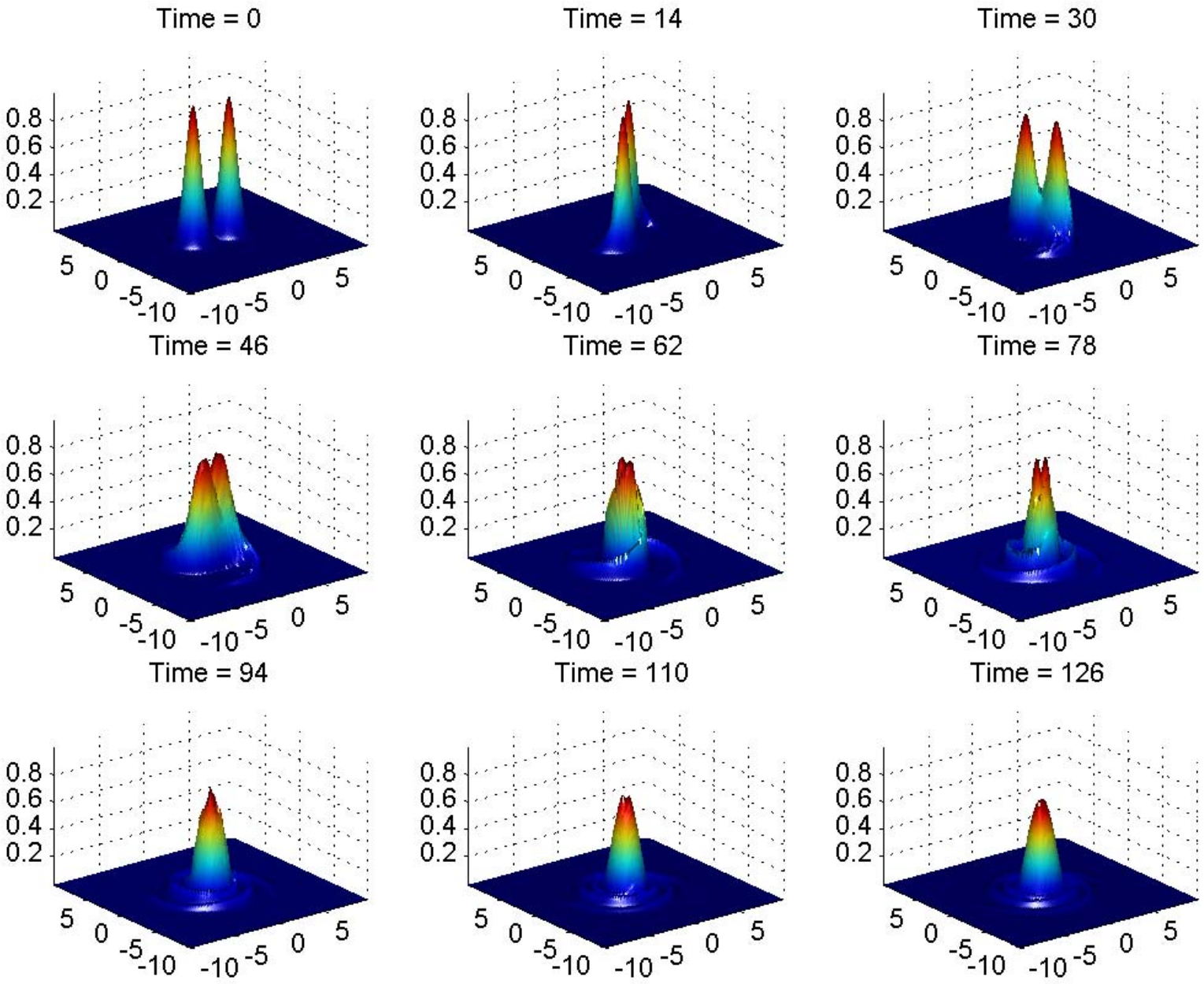


Figure 25: Two same Gaussians next to each other (using surf function)

This is an interesting simulation. Namely, two vortices actually merge together and form a single vortex. Once again, we can calculate the vorticity coefficient as a function of time to see how the overall vorticity evolves. This is shown below in Figure 26.

This is a particularly interesting result. It appears that the vorticity actually increases initially when the two vortices merge together. The maximum vorticity appears to occur at roughly t = 80 seconds. By referring to Figure 25, this is the point where the two vortices have just finished merging together. After this, the overall vorticity begins to drop. The most interesting part of this is that the overall vorticity over the computational domain is preserved and furthermore, this does not go to zero as time increases. In fact, it appears that the vorticity will become constant. One might ask, does this mean that there is no viscous diffusion? In essence, the answer is yes. By examining the advection/diffusion equation (Equation 2), it can be seen that diffusion only occurs if $\nabla^2\omega \neq 0$. Recall that

$$\nabla^2\omega = \frac{\partial^2\omega}{\partial x^2} + \frac{\partial^2\omega}{\partial y^2}$$

Therefore, one can see that this term can become zero if there is no change in vorticity in the x or y directions. In this situation, this is exactly what happens. Since the two Gaussians are the same magnitude, after they merge together, there is no acceleration of vorticity in either the x or y direction, so the diffusion terms becomes zero. This means that there is no diffusion effect so the vorticity is maintained at the present level indefinitely. Therefore, one can draw the conclusion that the diffusion effect only takes place when there is changes in vorticity with respect to the x and y directions. Therefore, the vorticity is not changing much after t = 100 seconds (when the vorticity coefficient is constant). This is shown below in Figure 27.

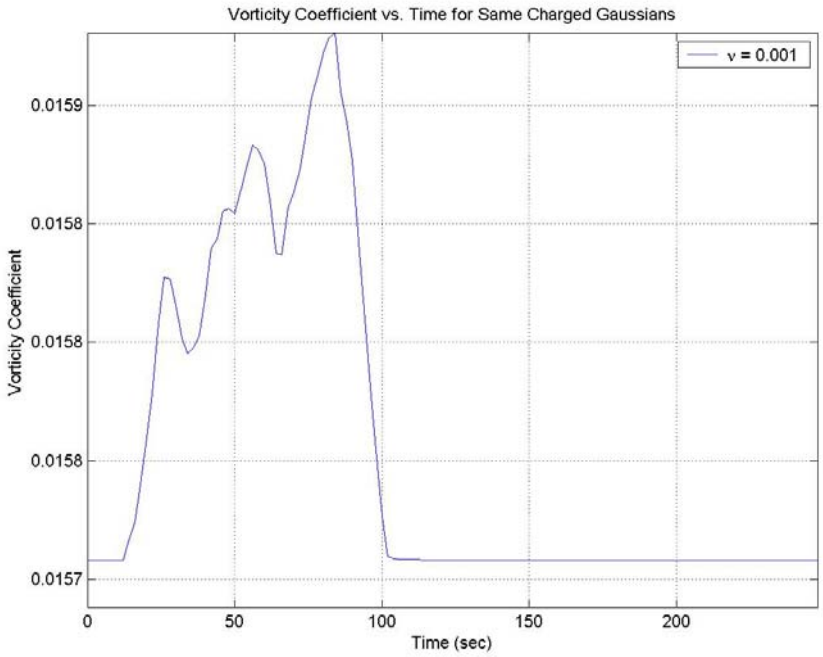


Figure 26: Vorticity coefficient vs. time for two same charged Gaussians

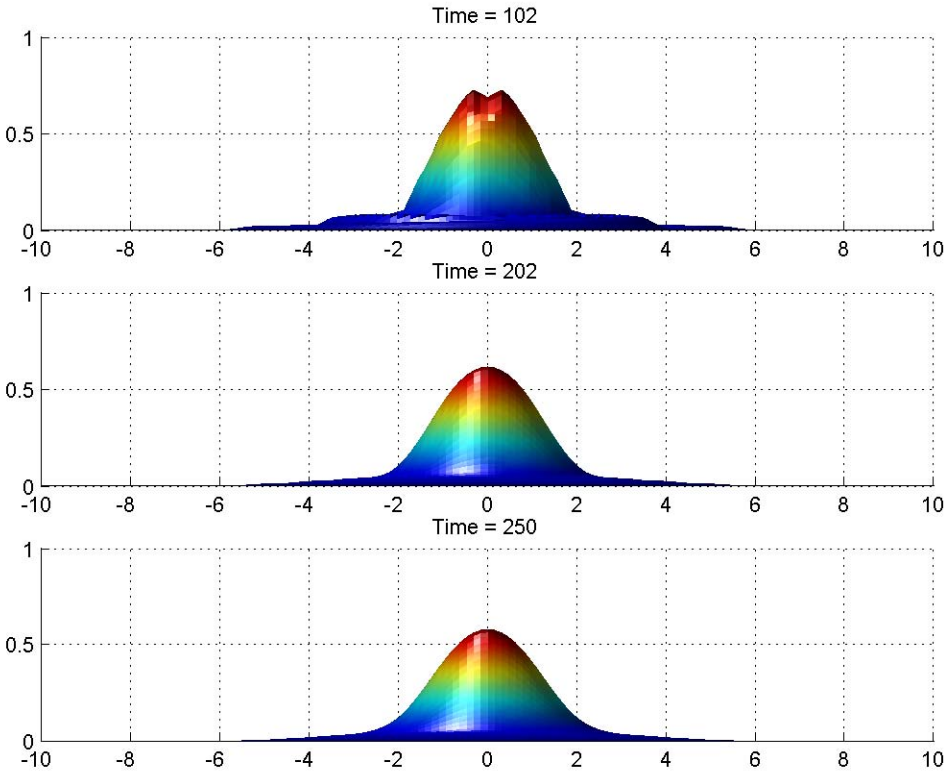


Figure 27: Vorticity after two Gaussians merge together

As can be seen, the vorticity is not changing much after t = 100. Furthermore, there is no change in vorticity with respect to the x or y direction. This explains why the vorticity coefficient is roughly constant after t = 100 seconds. If the vorticity is not changing rapidly, then the system states are not changing very rapidly. An interesting observation can be made by looking at an excerpt from the screen diagnostic printout shown below.

```
    Solution Process
    10% Complete.    42.891 sec elapsed
    20% Complete.    112.938 sec elapsed
    30% Complete.    198.875 sec elapsed
    40% Complete.    241.016 sec elapsed
    50% Complete.    255.031 sec elapsed
    60% Complete.    263.235 sec elapsed
    70% Complete.    271.719 sec elapsed
    80% Complete.    279.906 sec elapsed
    90% Complete.    287.875 sec elapsed

Solution complete!

    126 time intervals written to variable
    Time to solve ode call = 297.063 sec.

TOTAL TIME = 297.5 sec.

---------------------------------
--PRESS ANY KEY TO PLAY SOLUTION--
---------------------------------
```

After t = 100 (roughly 40% complete), the vorticity has been shown to not change rapidly. A plot of percentage complete vs. total time elapsed is shown below in Figure 28.

As can be seen, the solution progresses much faster after 40% complete, which corresponds to the time when the two vortices converge. After they converge, the vorticity is not changing very fast and therefore, the ode solver can afford to take larger time steps, thereby making the solution progress faster.

As shown previously, the vorticity coefficient is only decreased when there is a change in vorticity with respect to the x and y direction. If there is a lot of motion (two opposite Gaussians), then the vorticity should decrease over time. This will be shown in the next example, where two pairs of opposite vortices are made to collide with each other. The initial condition for this situation is shown below in Figure 29.

The evolution of this system as a function of time is shown below in Figure 30 and Figure 31.

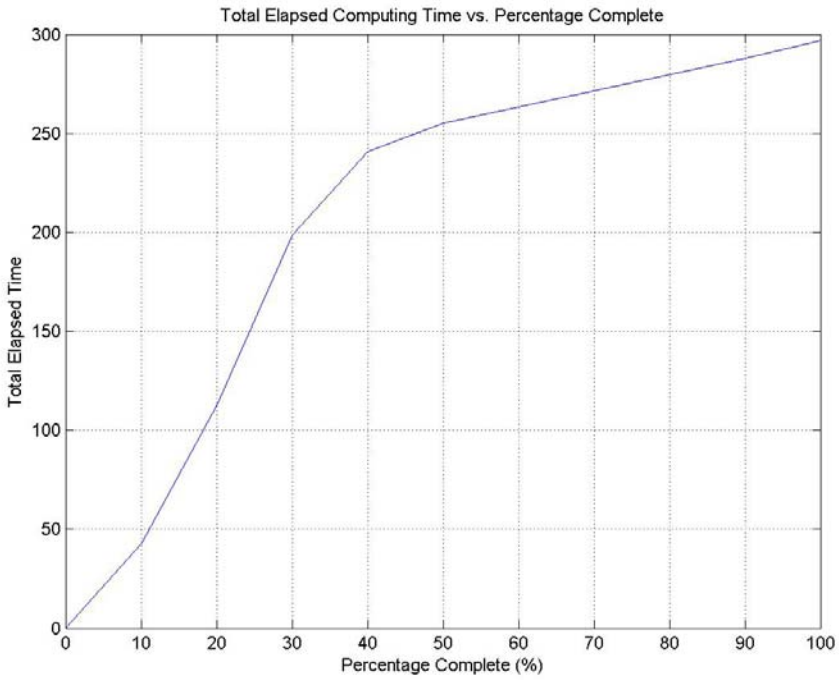


Figure 28: Total computational time elapsed vs. percentage complete for two same Gaussians

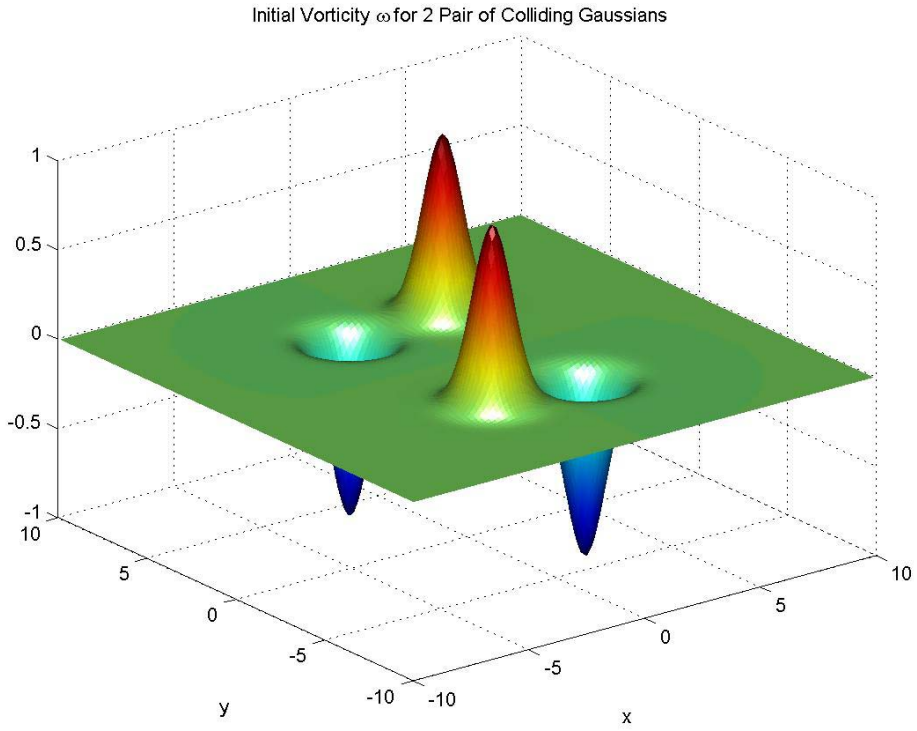


Figure 29: Initial vorticity for two pairs of colliding Gaussians

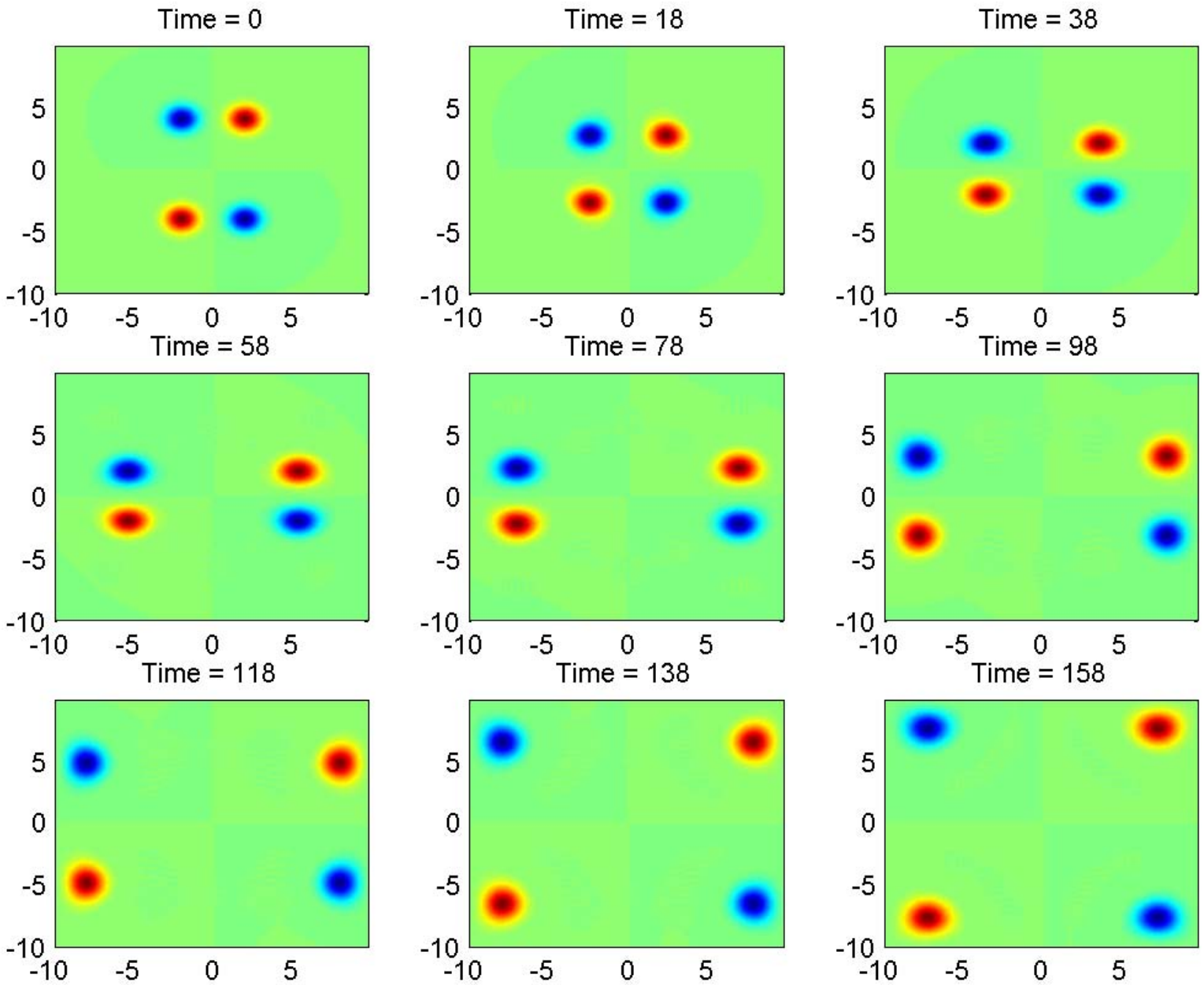


Figure 30: Two pairs of oppositely charged Gaussians next to each other (using pcolor function)

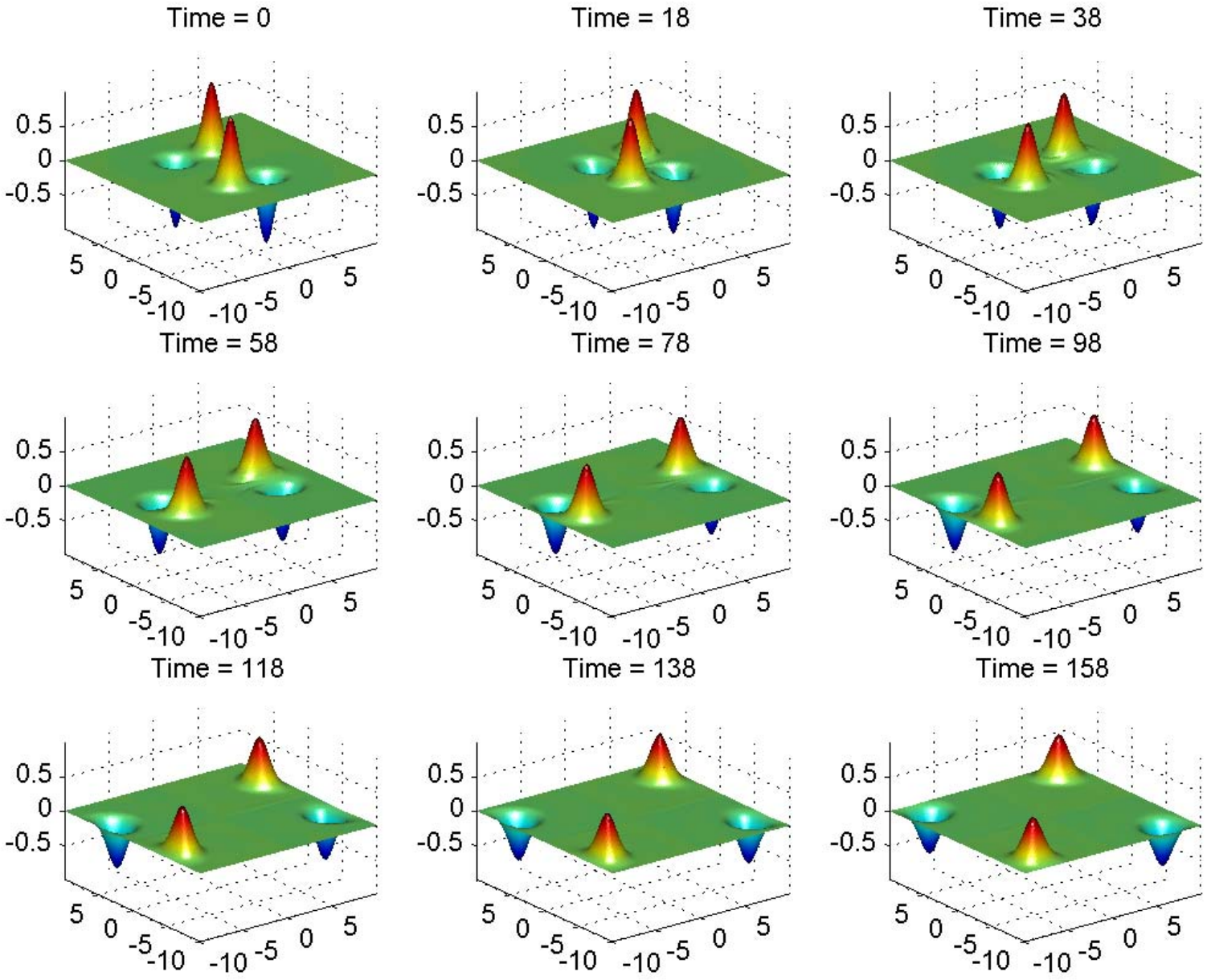


Figure 31: Two pairs of oppositely charged Gaussians next to each other (using pcolor function)

This simulation produces some very interesting results. Namely, the two pairs of vortices move together until they collide. After they collide, the break apart and move together with the other vortex. One can also see the effects of the periodic boundary conditions near the end of the simulation as the vortex pairs move since they are affected by the other vortex across the boundary. The theory that the diffusion effect only takes place with moving vortices can be tested by plotting the vorticity coefficient below in Figure 32.

As can be seen, the overall vorticity does decrease dramatically with two pairs of oppositely charged Gaussians. This makes sense considering that these are moving pairs of vortices and they continue to create changes in vorticity with respect to the x and y direction.

In the previous two examples, it has been shown that vorticity is increased when vortices collide and vorticity decreases if the vortices move in the x and y direction. We can also perform a simulation where both movement and combinations of vortices are present. The initial condition for this situation is shown below in Figure 33.

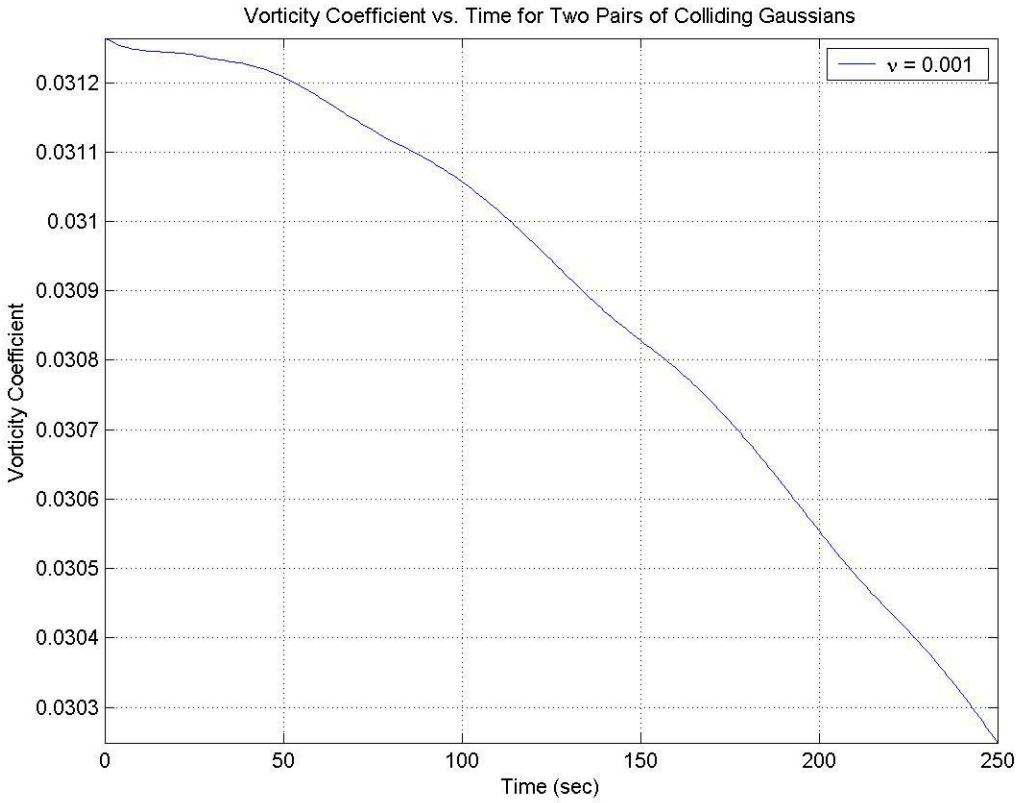


Figure 32: Vorticity coefficient vs. time for two pairs of oppositely charged Gaussians

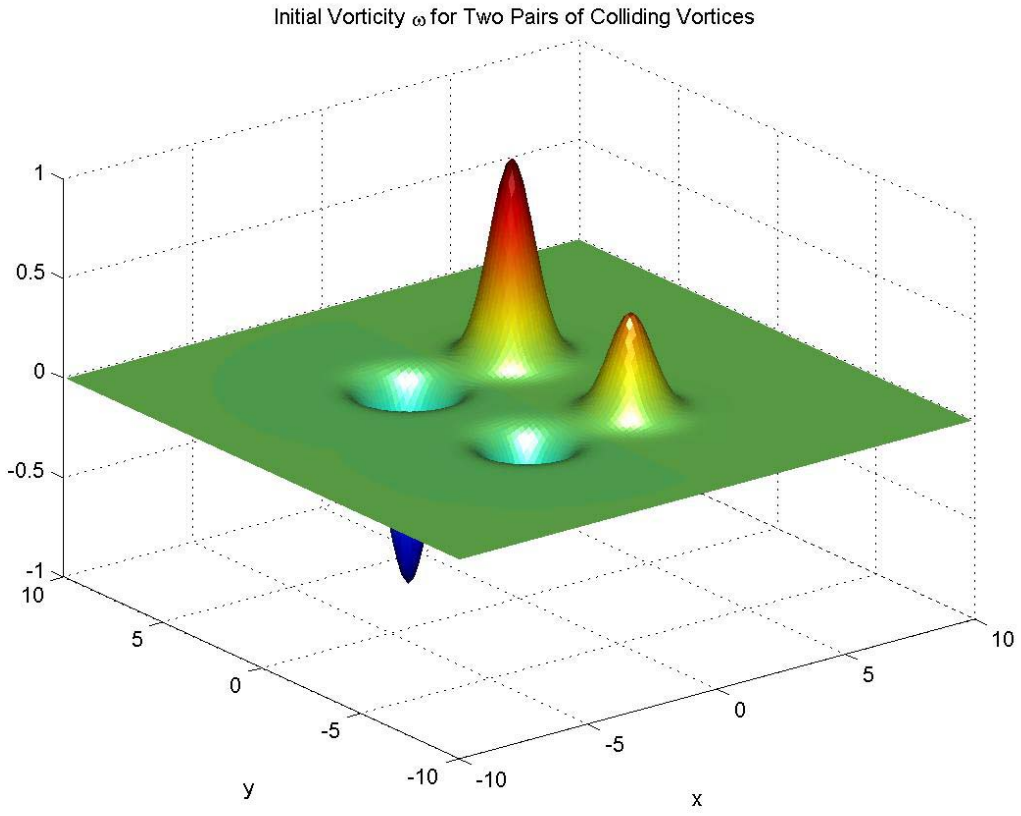


Figure 33: Initial vorticity for two pairs of colliding Gaussians

This system is two pairs of Gaussian vortices. The pair with the higher amplitude should move faster than the pair with smaller amplitude. This in turn means that the larger vortex pair will eventually catch up and collide with the smaller vortex pair. Furthermore, since the vortices of like amplitude will collide, this will yield a combination of vortices. The evolution of this system is shown below in Figure 34 and Figure 35.

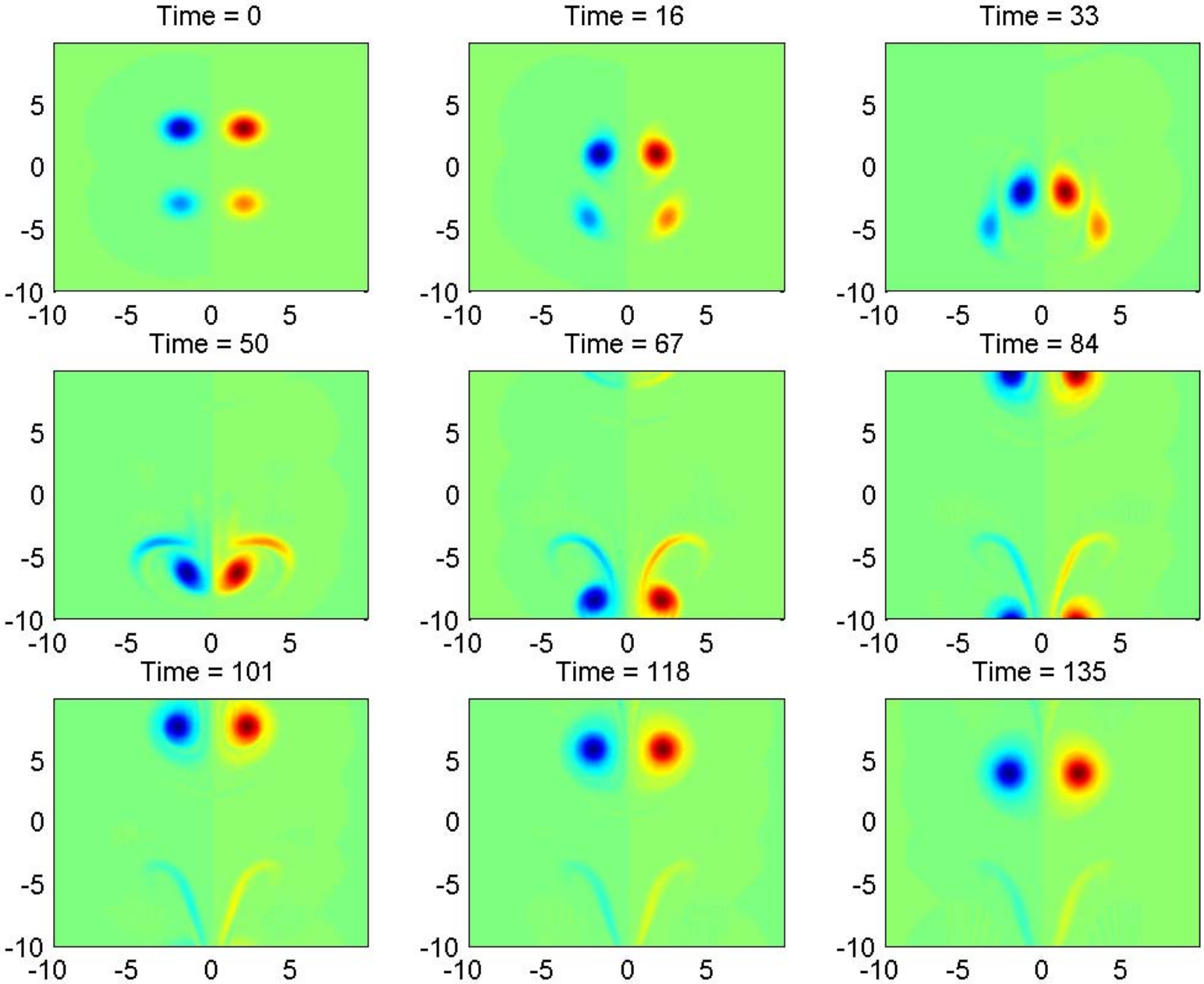


Figure 34: Two pairs of oppositely charged Gaussians next to each other (using pcolor function)

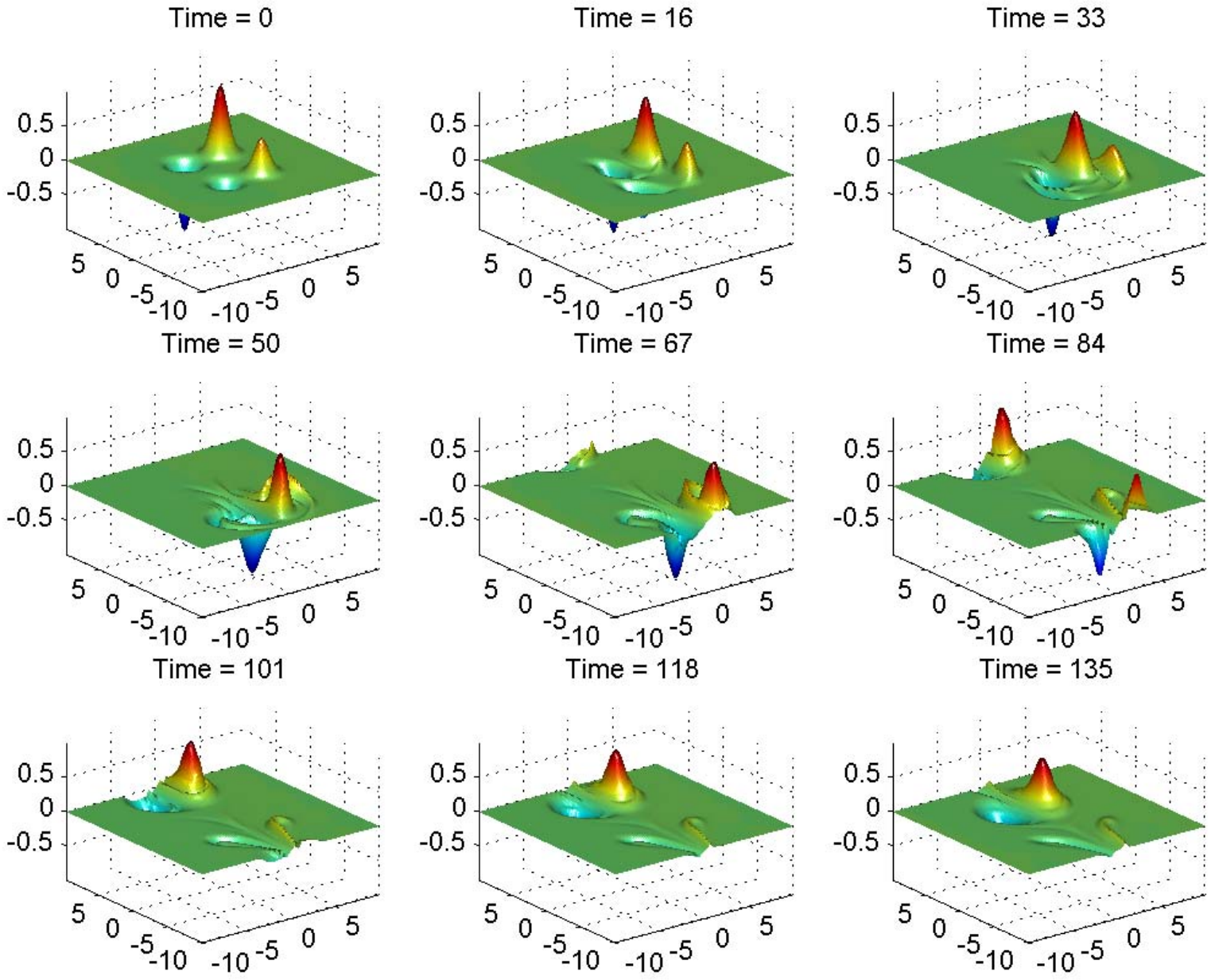


Figure 35: Two pairs of oppositely charged Gaussians next to each other (using surf function)

As can be seen, the larger vortex pair catches up with the smaller pair and consumes it. According to the theory above, the vorticity coefficient should decrease as the vortices move, but it should increase when the combine. A plot of the vorticity coefficient as a function of time is shown below in Figure 36.

From Figure 36 one can see that this does seem to fit with the theory. The time around t = 60 is when the vortices are colliding, so there is a rise in vorticity. However, since the both are still moving, the diffusion still serves to decrease the vorticity, so the vorticity coefficient continues to drop.

The last simulation to run is simply a series of random vortices. One precaution for this situation is that all of the random vortices must be initially contained well within the computational domain. Due to the periodic boundary conditions, it is possible to start a vortex half way outside the boundary and the solution will progress. This can be seen by shifting the two opposite Gaussians (both of magnitude 1), half way down the computational domain. The initial condition for this situation is shown below in Figure 37.

As can be seen, this is not a realistic simulation. This is the same initial condition presented previously, except the vortices are shifted down so that half of it is inside the computational domain. The progression of this initial condition vs. time is shown below in Figure 38.

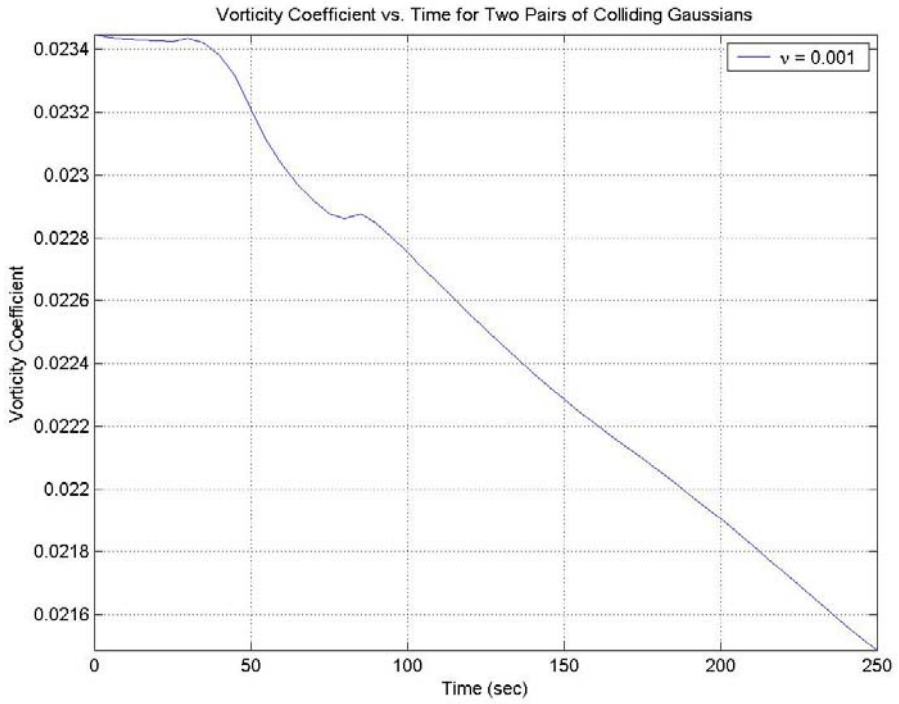


Figure 36: Vorticity coefficient vs. time for two pairs of oppositely charged Gaussians

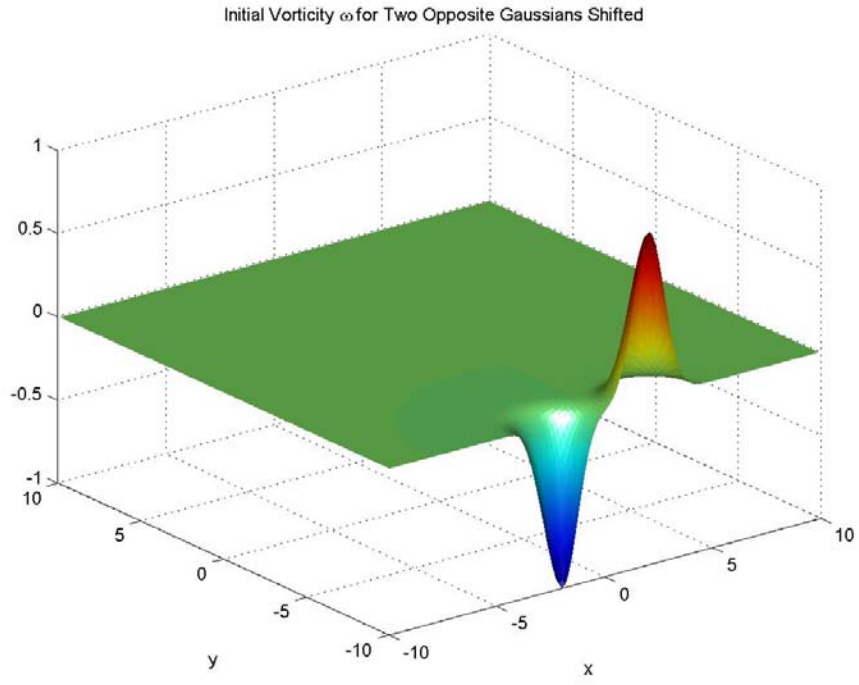


Figure 37: Initial condition for two vortices shifted down computational domain.

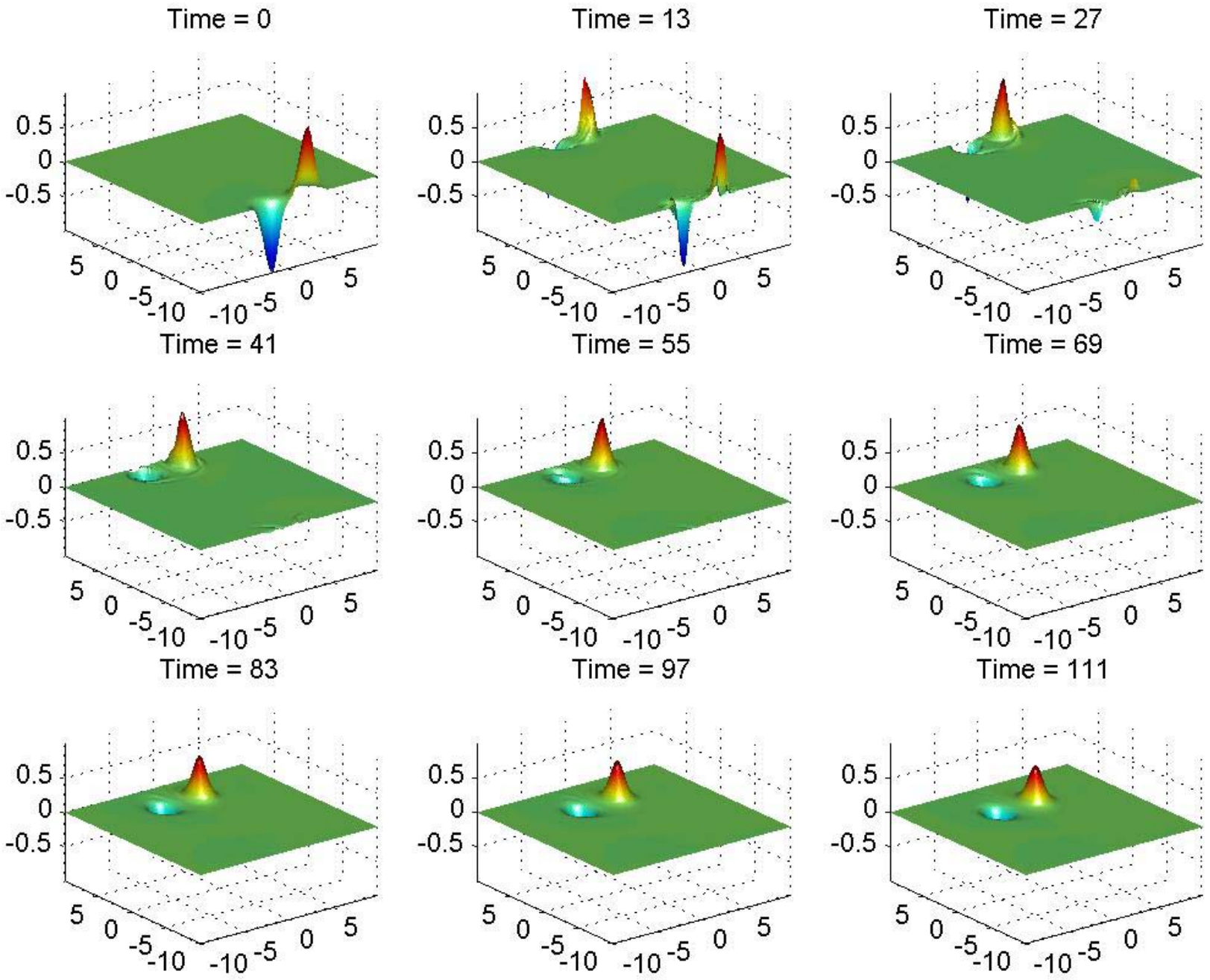


Figure 38: Two opposite Gaussian shifted (using surf function)

As can be seen, the periodic boundary conditions are able to deal with this discontinuity, but it takes a while for the simulation to settle down. Also, since there is such a sharp discontinuity between the sides of the computational domain, the states are changing very rapidly initially. This causes the solution to progress slowly at first and then speed up once the vorticity has settled down and the states are not moving very fast.

Once all of the random vortices are contained well within the computational domain, the simulation can be performed. The initial condition for this situation is shown below in Figure 39.

As can be seen, all of the vortices are well contained within the computational domain. The evolution of this system is shown below in Figure 40 and Figure 41.

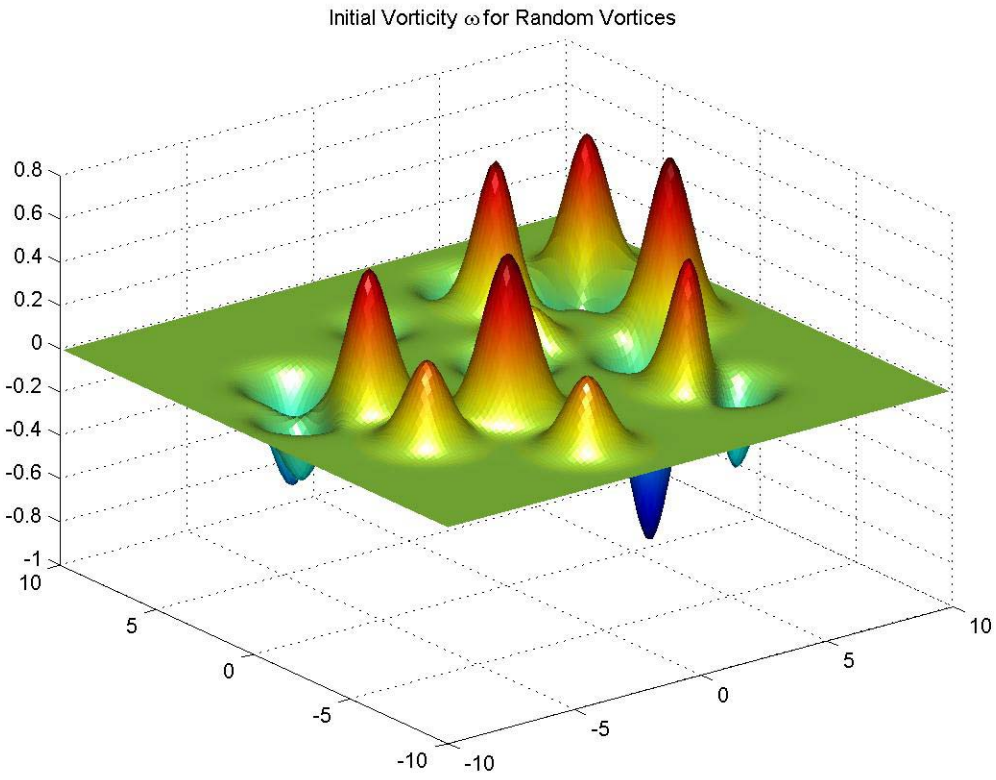


Figure 39: Initial conditions for random vortices

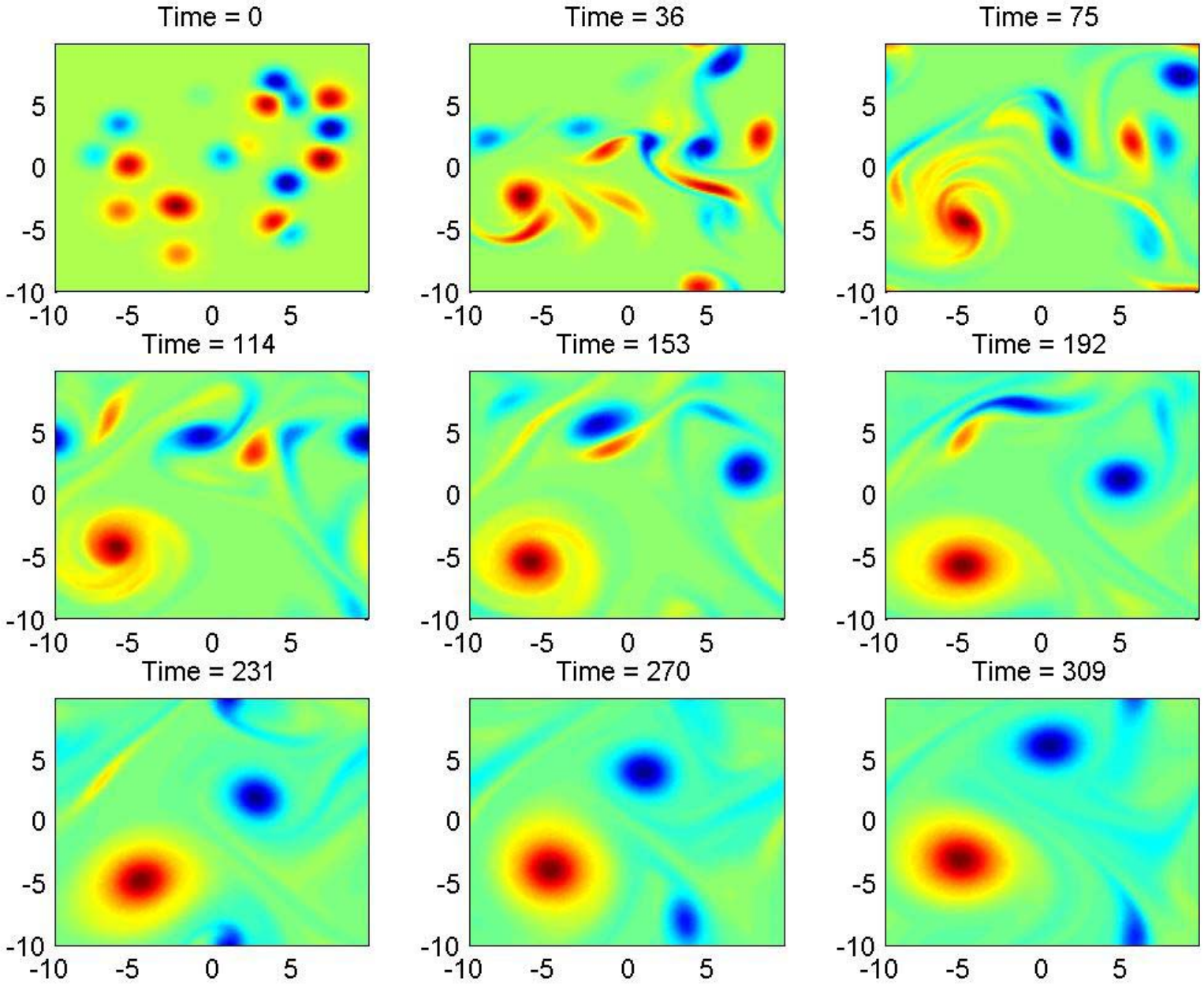


Figure 40: Random vortices simulation (using pcolor)

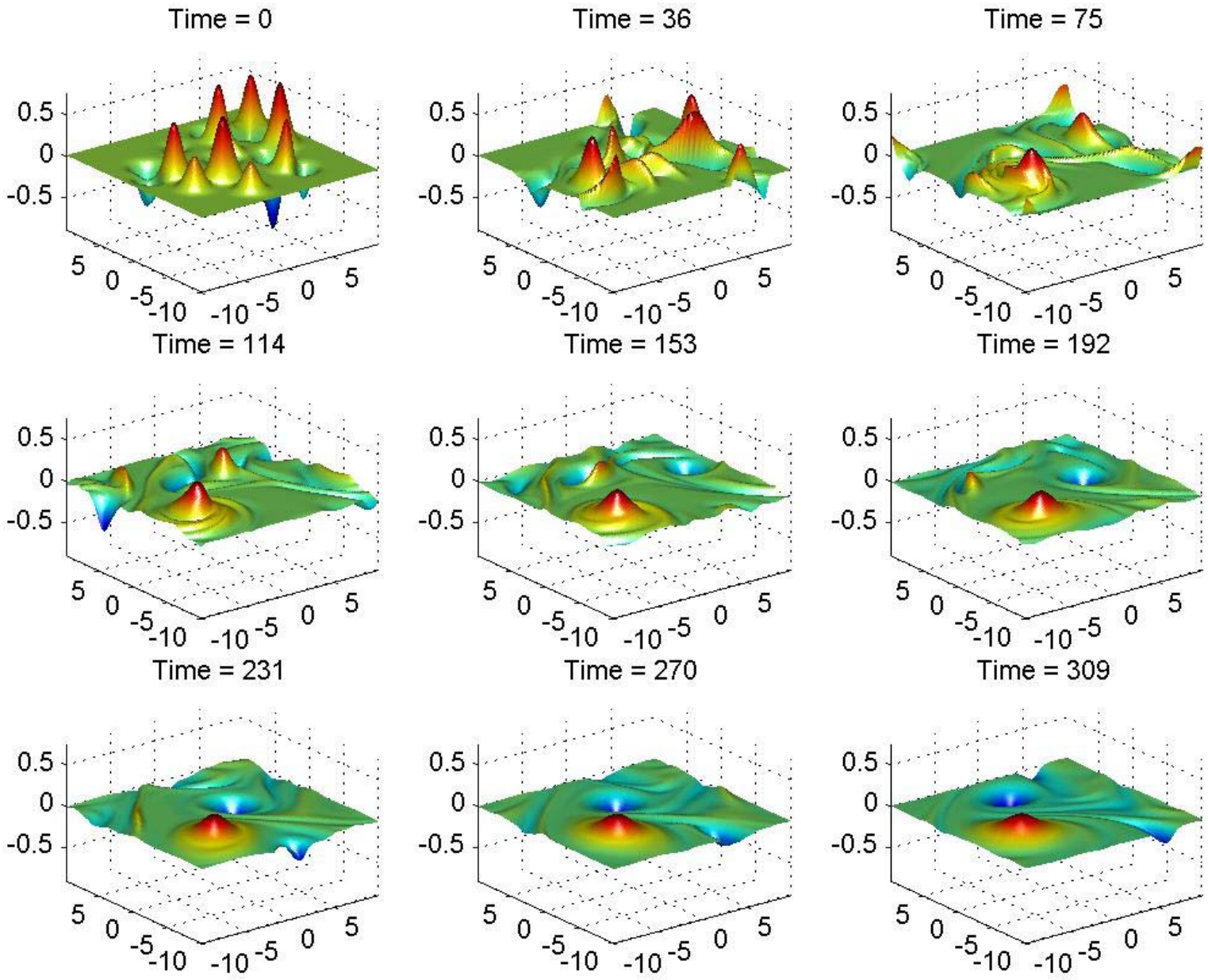


Figure 41: Random vortices simulation (using surf function)

The results here are a combination of conclusions from all the previous sections. Namely, opposite vortices move together and similar vortices combine to create a single vortex. All of these vortices move around and combine until there is only a single pair of vortices that moves along. As time increases, the diffusion coefficient decreases the overall vorticity. This is shown below in Figure 42.

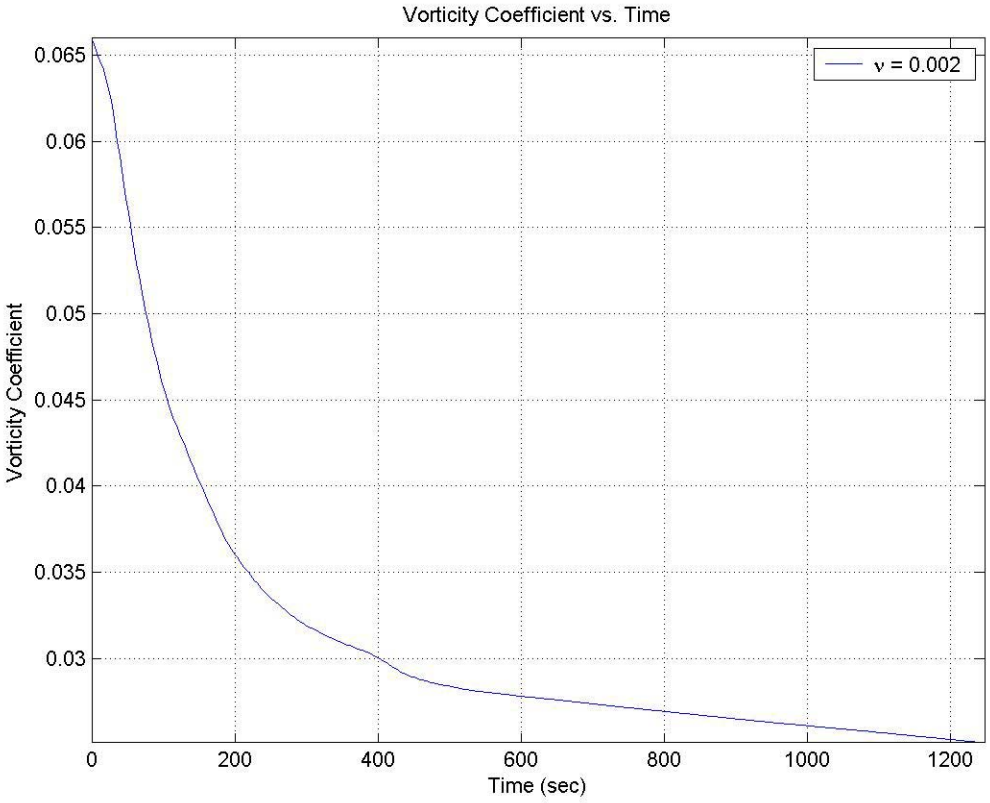


Figure 42: Vorticity coefficient vs. time for random vortices

Conclusion

The results of this simulation are very encouraging. It appears that this produces realistic results and provides a basis for more complex computation fluid dynamic models. It proves that discretization and numerical methods are viable techniques to solve the governing partial differential equations.

One of the most useful tools for analyzing the results was the vorticity coefficient. Since it was derived independently, there has not been much of a chance to test it for accuracy and robustness. Given more time, more work with the vorticity coefficient would be done. Specifically, more research would be directed towards understanding why the vorticity appears to increase when vortices are combined. Conventional wisdom (and Kelvin's Circulation Theory) might dictate that this appears to be a conservative force field where energy (or vorticity) is conserved and the diffusion coefficient is the only agent to decrease the vorticity.

Another future research area would be to investigate the actual operation cost of each elliptic solver in terms of number of operations instead of time required. Since MatLab 6.5 does not allow access to the number of flops, or operations performed, this could be done by first calibrating the speed of the computer. Namely, timing how long it takes the computer to perform a known number of operations. This could then be used to calculate the number of operations per second. The `cputime` function could then be used to time the elliptic solver and the number of operations could be backed out.

A final investigation that would be performed with more time would be a closer look into the spectral implementation to solve the problem. Theoretically, it is not necessary to shift the vorticity back into the spatial domain before passing it out of the ode call. It should be possible to use ode23 to solve for the vorticity transformed in x and y. However, MatLab does not calculate the Fourier transform correctly. Instead, the transform has a larger amplitude than the analytical result. This amplitude magnification appears to be a function of number of wave numbers as well. This causes computational difficulties because this amplification causes ode23 to think that the states are changing much faster than the actually are. To compensate for this, it takes very small time steps, and therefore, the total time required to solve the problem suffers.

Appendix A: MatLab Functions

`[T,Y] = ODE23(ODEFUN,TSPAN,Y0,OPTIONS,P1,P2,…)`

`ODE23` performs numerical integration of a set of first order differential equations defined in the function, `ODEFUN`. It performs this integration over the interval `TSPAN` with initial conditions `Y0`. The error tolerance and other options can be changed with the `OPTIONS` variable, which is set using the `ODESET` function. Parameters `P1`, `P2`, etc. which are needed by the function in `ODEFUN` can be passed as well.

`A = SPDIAGS([e1 e2],-m m,n,n)`

The `SPDIAGS` function creates a sparse matrix with diagonals, `e1` and `e2`, etc. – `m` and `m` rows below and above the diagonal, respectively, in an `n` by `n` matrix.

`[X,Y] = MESHGRID(x,y)`

`MESHGRID` transforms the n-dimensional vectors, `x` and `y`, into an `n` x `n` matrices, `X` and Y which can be used for evaluation of 3 dimensional functions.

`X_vec = RESHAPE(X,n,1)`

`RESHAPE` transforms the `sqrt(n)` by `sqrt(n)` matrix `X` into a `n` dimensional vector. This works the other direction as well, from a vector to a matrix.

`[L,U,P] = LU(A)`

LU performs LU decomposition on the matrix A to return the lower triangular matrix L, the upper triangular matrix U, and the perturbation matrix P.

`Wo_t = FFT2(Wo,KX,KY)`

`FFT2` calculates the 2 dimensional Fourier transform of `Wo` in the `x` and `y` direction with associated wave numbers `KX` and `KY`.

`x = A\b`

The backslash command (\ ) solves the matrix equation A x = b using different methods depending on the structure of A.

```
[x,FLAG,RELRES,ITER,RESVEC] =
GMRES(A,b,[],TOL,MAX_ITER,[],[],XO]
```

GMRES iteratively solves the matrix equation A x = b by iterating from an initial guess XO. This continues for MAX_ITER or until the tolerance TOL is achieved. It returns the solution x along with a FLAG which signals if the solution converged. It also returns the vector of iterations ITER (in half steps) and the relative residual RESVEC as a function of iteration number.

```
[x,FLAG,RELRES,ITER,RESVEC] =
BICGSTAB(A,b,[],TOL,MAX_ITER,[],[],XO]
```

See GMRES

```
PCOLOR(X,Y,Wo)
```

PCOLOR creates a top down view on a domain of X and Y of the 3 dimensional data with the magnitude of Wo at a given point represented by a color.

```
COLORBAR
```

COLORBAR creates a legend, which correlates the color of a PCOLOR plot with the associated magnitude.

```
SURF(X,Y,Wo)
```

SURF creates a three dimensional plot of the data Wo over a domain of X and Y.

```
t = cputime
```

cputime returns the current time and assigns it to the variable t.

Appendix B: MatLab Code

main.m

```
%Christopher Lum
%9923916
%AMATH581 Kutz
%Nov. 5, 2003
%
%HW2

clear
clc
close all

%Declare global variables
global trigger
global ode_t
global process_time

%Obtain user preferences
[solution_method,matrix_selection,vorticity_type,vorticity_selection,...
        fft_selection,relres_selection,camera_options,wave_selection,...
        streamfunction_selection,L,m,v,num_relres_plots,t_final,psi_update,...
        save_data,max_iter] = user_preferences;

%Extract camera options
az_start = camera_options(1);
az_end = camera_options(2);
el_start = camera_options(3);
el_end = camera_options(4);

%Calculate parameters
n = m^2;                        %Total size of matrix

xyspan2 = linspace(-L/2,L/2,m+1);
xyspan = xyspan2(1:m);

%Calculate delta = dx = dy
delta = xyspan(2) - xyspan(1);

%BICGSTAB and GMRES parameters.  Tolerance should be consistent with accuracy of
%discretization
tol = delta^2*0.75;

resvec_data = 0;                %Initialize matrix to store relative residual data.
psio_col_data = zeros(1,n);     %Initialize matrix to store streamfunction.

%Initialize global variable trigger and process_time with all zeros.
trigger = zeros(10,1);
process_time = zeros(11,1);

tic

%-------------------------Generate Matrices-------------------------
[A,B,C] = create_matrices(n,m,delta,matrix_selection);


%------------------Create Initial Vorticity-------------------------
[X,Y,wo] = create_vorticity(xyspan,vorticity_type,vorticity_selection,L);

%-----------------Streamfunction Updates--------------------------
%How intervals does this mean we have to solve over
num_intervals = floor(t_final/psi_update);


%-----------------Pick Elliptic Solver----------------------------
```

```
switch solution_method
    case 1
        %----------------------------FFT--------------------------
        disp('Now solving using FFT...')
        disp(' ')

        %Define integer wave numbers
        kx = (2*pi/L)*[0:(m/2-1) (-m/2):-1];
        ky = kx;

        %Set the first element of each to a small number
        kx(1) = 10e-6;
        ky(1) = 10e-6;

        [KX,KY] = meshgrid(kx,ky);

        %Are we analyzing the wave numbers?
        if wave_selection==1
            analyze_wave_function(xyspan,kx)
        else
        end


        %Which method should we use to update the streamfunction?
        if fft_selection==1
            %UPDATE AT EVERY STEP
            disp('Updating streamfunction at every step.')
            disp(' ')
            disp('Please wait...')
            disp('And while you''re waiting, here''s a little riddle...')

            %Ask riddle
            ask_question;

            %Print parameters being used
            display_parameters(L,m,v,vorticity_type,t_final,psi_update,save_data)

            %Reshape initial vorticity.
            wo_col = reshape(wo,n,1);

            ode_t = cputime;

            [t,w_col] = ode23('advection_fft',[0:psi_update:t_final],wo_col,...
                [],A,B,C,v,KX,KY,m,t_final);

            %What is the time to complete the whole simulation?
            process_time(11) = cputime - ode_t;

            disp(' ')
            disp('Solution complete!')

            disp(' ')
            disp(['    ',num2str(length(t)),' time intervals written to variable'])
            disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
            disp(' ')

            disp(['TOTAL TIME = ',num2str(toc),' sec.'])
            w_col = real(w_col);

            %Do we want to save data?
            if save_data==1
                save vortex_data w_col X Y m L v psio_col_data t process_time
            else
            end

            %Let's plot the vorticity
            disp(' ')
            disp('-----------------------------------')
            disp('--PRESS ANY KEY TO PLAY SOLUTION--')
            disp('-----------------------------------')
            pause
```

```
            disp(' ')

            %create an array for camera rotation.
            az_array = linspace(az_start,az_end,length(t));

            el_array = linspace(el_start,el_end,length(t));

            figure
            for counter=1:length(t)
                w = reshape(w_col(counter,:),m,m);
                surf(X,Y,w)
                title(['Time = ',num2str(round(t(counter)))])

                %What is current view?
                view(az_array(counter),el_array(counter));
                drawnow
            end


        else

            %UPDATE ONLY AT DISCRETE STEPS
            disp('Updating only at discrete steps.')
            disp(' ')

            az_array = linspace(az_start,az_end,num_intervals);

            el_array = linspace(el_start,el_end,num_intervals);

            figure
            for counter=1:num_intervals

                section_t= cputime;
                elliptic_t = cputime;

                disp(['Solving for t = ',num2str(psi_update*(counter-1)),...
                        ' sec to t = ',num2str(psi_update*(counter))])

                %Calculate the FFT of the vorticity
                wo_t = fft2(wo);

                %Solve del^2*psi = w in fourier domain
                psio_t = (-wo_t)./(KX.^2 + KY.^2);

                %Shift psi back into spatial domain
                psio = ifft2(psio_t);

                %We need to reshape the voticity and stream function matrices into
                %columns
                wo_col = reshape(wo,n,1);
                psio_col = reshape(psio,n,1);

                %Do we want to save the data so we can plot both the
                %streamfunction and the vorticity?
                if streamfunction_selection==1
                    wo_col_data(counter,:) = wo_col';
                    psio_col_data(counter,:) = real(psio_col');

                    %What is the total time elapsed?
                    t_data(counter) = (counter-1)*psi_update;
                else
                end

                disp(['    Time for elliptic solve w/ FFT = ',num2str(cputime -
elliptic_t),' sec.'])

                %Solve advection/diffusion eq. to advance vorticity into future
                %w/ constant streamfunction.
                ode_t = cputime;

                [t,wo_col] = ode23('advection',[0 psi_update],wo_col,...
```

```
                    [],A,B,C,psio_col,v);

                disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
                disp(['    ',num2str(length(t)),' time steps used for ode call.'])

                wo_col = real(wo_col);

                %Let's look at the vorticity
                wo = reshape(wo_col(end,:),m,m);
                surf(X,Y,wo)
                drawnow
                view(az_array(counter),el_array(counter));

                disp(' ')
                disp(['    Section time = ',num2str(cputime - section_t),' sec.'])
                disp(' ')
            end
        end

        disp(['TOTAL TIME = ',num2str(toc),' sec.'])



    case 2
        %-----------------------------A\b---------------------------------
        disp('Now solving using A\b...')
        disp(' ')

        %Create a non-singular A matrix
        A2 = A;
        A2(1,1) = 0;

        figure
        for counter=1:num_intervals

            section_t = cputime;
            elliptic_t = cputime;

            disp(['Solving for t = ',num2str(psi_update*(counter-1)),...
                    ' sec to t = ',num2str(psi_update*(counter))])

            %Need to reshape the initial vorticity
            wo_col = reshape(wo,n,1);

            %Solve for streamfunction.
            psio_col = A2\wo_col;

            disp(['    Time for elliptic solve with A\b = ',num2str(cputime -
elliptic_t),' sec.'])

            ode_t = cputime;

            [t,wo_col] = ode23('advection',[0 psi_update],wo_col,...
                [],A,B,C,psio_col,v);

            disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
            disp(['    ',num2str(length(t)),' time steps used for ode call.'])

            wo_col = real(wo_col);

            %Let's look at the vorticity
            wo = reshape(wo_col(end,:),m,m);
            pcolor(X,Y,wo)
            drawnow

            disp(' ')
            disp(['    Section time = ',num2str(cputime - section_t),' sec.'])
            disp(' ')
        end

        disp(['TOTAL TIME = ',num2str(toc),' sec.'])
```

```
    case 3
        %---------------------LU Decomposition-----------------------------
        disp('Now solving using LU Decomposition...')
        disp(' ')

        %Perform LU decomposition, O(N^3) once.  We need to do this with a
        %non-singular A matrix but we cannot have the frist value be a 0 or
        %else the pivot point will have the change and the permutation
        %matrix will not be the identity matrix.
        A2 = A;
        A2(1,1) = 1;

        factorization_t = cputime;
        [Lmatrix,Umatrix,P] = lu(A2);

        disp(['TIME TO OBTAIN L AND U = ',num2str(cputime - factorization_t),...
                ' sec.'])
        disp(' ')

        figure
        for counter=1:num_intervals

            section_t = cputime;
            elliptic_t = cputime;

            disp(['Solving for t = ',num2str(psi_update*(counter-1)),...
                    ' sec to t = ',num2str(psi_update*(counter))])

            %Need to reshape the initial vorticity
            wo_col = reshape(wo,n,1);

            %Solve for streamfunction using LU decomposition
            forward_t = cputime;
            y = Lmatrix\wo_col;

            disp(['    Time for forward substitution = ',num2str(cputime - forward_t),...
                    ' sec.'])

            backward_t = cputime;
            psio_col = Umatrix\y;

            disp(['    Time for backward substitution = ',num2str(cputime -
backward_t),...
                    ' sec.'])

            disp(['    Time for elliptic solve with LU = ',num2str(cputime -
elliptic_t),...
                    ' sec.'])

            ode_t = cputime;

            [t,wo_col] = ode23('advection',[0 psi_update],wo_col,...
                []',A,B,C,psio_col,v);

            disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
            disp(['    ',num2str(length(t)),' time steps used for ode call.'])

            wo_col = real(wo_col);

            %Let's look at the vorticity
            wo = reshape(wo_col(end,:),m,m);
            pcolor(X,Y,wo)
            drawnow

            disp(' ')
            disp(['    Section time = ',num2str(cputime - section_t),' sec.'])
            disp(' ')
        end
```

```
    disp(['TOTAL TIME = ',num2str(toc),' sec.'])


case 4
    %-----------------------------BICGSTAB-------------------------------
    disp('Now solving using BICGSTAB...')
    disp(' ')

    %Create a non-singular A matrix
    A2 = A;
    A2(1,1) = 0;

    %Initialize a guess for bicgstab
    psio_col = zeros(n,1);

    figure
    for counter=1:num_intervals

        section_t = cputime;
        elliptic_t = cputime;

        disp(['Solving for t = ',num2str(psi_update*(counter-1)),...
                ' sec to t = ',num2str(psi_update*(counter))])

        %Need to reshape the initial vorticity
        wo_col = reshape(wo,n,1);

        %Solve for streamfunction using BICGSTAB
        [psio_col,flag,relres,iter,resvec] = bicgstab(A2,wo_col,...
            tol,max_iter,[],[],psio_col);

        %Check to see if it converged
        switch flag
            case 0
                %bicgstab converged.
                if relres_selection==1

                    %If we are interested in plotting the relative
                    %residual, we need to store the data.  We need to
                    %be careful since there is no gaurantee that the
                    %resvec array with be the same length each time
                    %bicgstab solves.  We need to be able to either
                    %stretch the matrix (if length(resvec) > length
                    %(resvec_data) or fill it with zeros (if
                    %length(resvec) < length(resvec_data).
                    iter_data(counter) = iter;

                    [resvec_rows,resvec_cols] = size(resvec_data);

                    length_resvec = length(resvec);     %No. rows of solution

                    if length_resvec > resvec_rows
                        %This means that this solution has more
                        %elements than the rest of the resvec_data
                        %matrix.  We need to strech the matrix and add
                        %zeros in all the other rows.
                        resvec_data(resvec_rows+1:length_resvec,:) = 0;
                    elseif length_resvec < resvec_rows
                        %This mean that the solution has less elements
                        %than the current matrix, this means that we
                        %need to add on zeros to the solution so that
                        %the dimension fits
                        resvec(length_resvec+1:resvec_rows,1) = 0;
                    else
                        %The solution will fit in the matrix, so do
                        %nothing
                    end

                    %Save the data
                    resvec_data(:,counter) = resvec;
                else
```

```
                    end

                case 1
                    disp('BICGSTAB DID NOT CONVERGE!!!')
                    pause(2)
                case 2
                    disp('Preconditioner M was ill-conditioned.')
                    pause(2)
                case 3
                    disp('bicgstab stagnated. (Two consecutive iterates were the same.)')
                    pause(2)
                case 4
                    disp('One of the scalar quantities calculated during bicgstab became
too small or too large to continue computing.')
                    pause(2)
                otherwise

            end

            disp(['    ',num2str(iter),' iterations required to converge to
',num2str(tol)])

            disp(['    Time for elliptic solve with BICGSTAB = ',num2str(cputime -
elliptic_t),...
                    ' sec.'])

            ode_t = cputime;

            [t,wo_col] = ode23('advection',[0 psi_update],wo_col,...
                [],A,B,C,psio_col,v);

            disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
            disp(['    ',num2str(length(t)),' time steps used for ode call.'])

            wo_col = real(wo_col);

            %Let's look at the vorticity
            wo = reshape(wo_col(end,:),m,m);
            pcolor(X,Y,wo)
            drawnow

            disp(' ')
            disp(['    Section time = ',num2str(cputime - section_t),' sec.'])
            disp(' ')
        end

        %Now check to see if we want to plot relative residuals
        if relres_selection==1
            for counter=1:num_relres_plots
                %What is the current iter value?
                current_iter = iter_data(counter);

                %What is the current resvec?
                current_resvec = resvec_data(:,counter);

                %We only want the non-zero entries
                current_resvec = current_resvec(find(current_resvec~=0));

                figure
                semilogy(0:0.5:current_iter,current_resvec)
                title(['BICGSTAB:  Iteration vs. Relative Residual for Elliptic Solve
#',...
                        num2str(counter)])
                xlabel('Iteration Number')
                ylabel('Relative Residual')
            end
        else
        end

        disp(['TOTAL TIME = ',num2str(toc),' sec.'])
```

```
case 5
    %----------------------------GMRES-------------------------------
    disp('Now solving using GMRES...')
    disp(' ')

    %Create a non-singular A matrix
    A2 = A;
    A2(1,1) = 0;

    %Initialize a guess for GMRES
    psio_col = zeros(n,1);

    figure
    for counter=1:num_intervals

        section_t = cputime;
        elliptic_t = cputime;

        disp(['Solving for t = ',num2str(psi_update*(counter-1)),...
                ' sec to t = ',num2str(psi_update*(counter))])

        %Need to reshape the initial vorticity
        wo_col = reshape(wo,n,1);

        %Solve for streamfunction using GMRES
        [psio_col,flag,relres,iter,resvec] = gmres(A2,wo_col,...
            [],tol,max_iter,[],[],psio_col);

        %Extract the inner and outer iteration counters
        iter_outer = iter(1);
        iter_inner = iter(2);

        %Check to see if it converged
        switch flag
            case 0
                %bicgstab converged.
                if relres_selection==1

                    %If we are interested in plotting the relative
                    %residual, we need to store the data.  We need to
                    %be careful since there is no gaurantee that the
                    %resvec array with be the same length each time
                    %bicgstab solves.  We need to be able to either
                    %stretch the matrix (if length(resvec) > length
                    %(resvec_data) or fill it with zeros (if
                    %length(resvec) < length(resvec_data).
                    iter_outer_data(counter) = iter_outer;
                    iter_inner_data(counter) = iter_inner;

                    [resvec_rows,resvec_cols] = size(resvec_data);

                    length_resvec = length(resvec);     %No. rows of solution

                    if length_resvec > resvec_rows
                        %This means that this solution has more
                        %elements than the rest of the resvec_data
                        %matrix.  We need to strech the matrix and add
                        %zeros in all the other rows.
                        resvec_data(resvec_rows+1:length_resvec,:) = 0;
                    elseif length_resvec < resvec_rows
                        %This mean that the solution has less elements
                        %than the current matrix, this means that we
                        %need to add on zeros to the solution so that
                        %the dimension fits
                        resvec(length_resvec+1:resvec_rows,1) = 0;
                    else
                        %The solution will fit in the matrix, so do
                        %nothing
                    end

                    %Save the data
```

```
                        resvec_data(:,counter) = resvec;
                    else
                    end

                case 1
                    disp('GMRES DID NOT CONVERGE!!!')
                    pause(2)
                case 2
                    disp('Preconditioner M was ill-conditioned.')
                    pause(2)
                case 3
                    disp('gmres stagnated. (Two consecutive iterates were the same.)')
                    pause(2)
                otherwise

            end

            disp(['    ',num2str(iter),' iterations required to converge to
',num2str(tol)])

            disp(['    Time for elliptic solve with GMRES = ',num2str(cputime -
elliptic_t),...
                    ' sec.'])

            ode_t = cputime;

            [t,wo_col] = ode23('advection',[0 psi_update],wo_col,...
                [],A,B,C,psio_col,v);

            disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
            disp(['    ',num2str(length(t)),' time steps used for ode call.'])

            wo_col = real(wo_col);

            %Let's look at the vorticity
            wo = reshape(wo_col(end,:),m,m);
            pcolor(X,Y,wo)
            drawnow

            disp(' ')
            disp(['    Section time = ',num2str(cputime - section_t),' sec.'])
            disp(' ')
        end

        %Now check to see if we want to plot relative residuals
        if relres_selection==1
            for counter=1:num_relres_plots
                %What is the current iter value?
                current_iter = iter_inner_data(counter);

                %What is the current resvec?
                current_resvec = resvec_data(:,counter);

                %We only want the non-zero entries
                current_resvec = current_resvec(find(current_resvec~=0));

                figure
                semilogy(0:current_iter,current_resvec)
                title(['GMRES:  Iteration vs. Relative Residual for Elliptic Solve #',...
                        num2str(counter)])
                xlabel('Iteration Number')
                ylabel('Relative Residual')
            end
        else
        end

        disp(['TOTAL TIME = ',num2str(toc),' sec.'])


    case 6
        %--------------------------Spectral------------------------------
```

```
        disp('Now solving using Spectral Implementation...')
        disp(' ')

        %Define integer wave numbers
        kx = (2*pi/L)*[0:(m/2-1) (-m/2):-1];
        ky = kx;

        %Set the first element of each to a small number
        kx(1) = 10e-6;
        ky(1) = 10e-6;

        [KX,KY] = meshgrid(kx,ky);


        %UPDATE AT EVERY STEP
        disp('Updating streamfunction at every step.')
        disp(' ')
        disp('Please wait...')
        disp('And while you''re waiting, here''s a little riddle...')

        ask_question;

        %Reshape initial vorticity.
        wo_col = reshape(wo,n,1);

        ode_t = cputime;
        [t,w_col] = ode23('advection_spectral',[0:psi_update:t_final],wo_col,...
            [],v,KX,KY,m,t_final);

        disp(' ')
        disp('Solution complete!')

        disp(' ')
        disp(['    ',num2str(length(t)),' points written to variable.'])
        disp(['    Time to solve ode call = ',num2str(cputime - ode_t),' sec.'])
        disp(' ')

        disp(['TOTAL TIME = ',num2str(toc),' sec.'])


        %Let's plot the vorticity
        disp(' ')
        disp('-----------------------------------')
        disp('--PRESS ANY KEY TO PLAY SOLUTION--')
        disp('-----------------------------------')
        pause
        disp(' ')

        %create an array for camera rotation.
        az_array = linspace(az_start,az_end,length(t));

        el_array = linspace(el_start,el_end,length(t));

        figure
        for counter=1:length(t)
            %Reshape transformed vorticity into a matrix
            w = reshape(w_col(counter,:),m,m);

            surf(X,Y,w)
            title(['Time = ',num2str(round(t(counter)))])

            %What is current view?
            view(az_array(counter),el_array(counter));

            drawnow
        end
    otherwise
        error('Invalid Elliptic Solve Method, Try Again')

end %End solution_method switch
```

user_preferences.m

```
function [solution_method,matrix_selection,vorticity_type,vorticity_selection,...
        fft_selection,relres_selection,camera_options,wave_selection,...
        streamfunction_selection,L,m,v,num_relres_plots,t_final,psi_update,...
        save_data,max_iter] = user_preferences

%Obtain the user preferences for the problem
%
%INPUT:     -None
%
%OUTPUT:    -Desired solution method
%           -Do you want to view matrices or not?
%           -What type of initial vorticity do you want?
%           -Do you want to view a plot of the initial vorticity?
%           -if using fft, do you want to update everytime or only at
%           discrete points?
%           -Preferences for bicgstab, gmres
%           -Preferences for camera rotation
%           -Preference on looking at wave numbers
%           -Preference on looking at streamfunction for FFT only
%           -Computational domain (L)
%           -Number of x and y directions (must be powers of 2) (m)
%           -Diffusion Coefficient (v)
%           -Number of relative residual plots
%           -Final time for simulation
%           -How often do you update streamfunction (psi_update)
%           -Save critical variables or not.

%Initialize variables
solution_method = 0;
matrix_selection = 0;
vorticity_type = 0;
vorticity_selection = 0;
fft_selection = 0;
camera_selection = 0;
relres_selection = 0;
wave_selection = 0;
streamfunction_selection = 0;
num_relres_plots = 0;
tol = 0;
max_iter = 0;

az_start = -37.5;
az_end = -37.5;
el_start = 30;
el_end = 30;

camera_options = [az_start az_end el_start el_end];

disp('Use saved settings or enter settings manually?')
disp(' ')
disp('  1 = Saved')
disp('  2 = Custom')
disp('  3 = Demonstration')
disp('  4 = Help')
disp(' ')
demo_selection = input('Please enter selection:  ');
clc

%Does the user want help?
if demo_selection==4
    disp('Settings can be saved by opening the m-file user_preferences.m')
    disp('User can then modify the parameters on line 81-97 so that')
    disp('concurrent simulations can be run without having to re-enter')
    disp('preferences.')
    disp(' ')
```

```
    disp('Rotating camera mode and saving is only for FFT with spectral implementation.')
    disp('Please see accompanying report for more information.')
    disp(' ')
    disp('    Created by:  Christopher Lum')
    disp('                 lum@u.washington.edu')

    disp(' ')
    disp('Use saved settings or enter settings manually?')
    disp(' ')
    disp('  1 = Saved')
    disp('  2 = Custom')
    disp('  3 = Demonstration')
    disp(' ')
    demo_selection = input('Please enter selection:  ');
    clc
else
end


if demo_selection==1
    %Saved settings
    solution_method = 1;                    %Desired solution method
    matrix_selection = 0;                   %View A, B, C matrices?
    vorticity_type = 5;                     %What type of initial condition?
    vorticity_selection = 0;                %View initial condition?
    fft_selection = 1;                      %If using fft, update at every time step?
    camera_options = [-37.5 -37.5 30 30];   %Camera options
    relres_selection = 0;                   %If using gmres/bicgstab, do you want plots?
    wave_selection = 0;                     %If using fft, do you want wavenumber plots?
    streamfunction_selection = 0;           %If using fft, non-spectral, view psi vs. t?
    L = 20;                                 %Computational domain
    m = 2^7;                                %Number of discretizations in x and y
    v = 0.002;                              %Diffusion coefficient
    num_relres_plots = 3;                   %If using gmres/bicgstab, number of plots.
    t_final = 120;                         %Final time of simulation
    psi_update = 3;                         %How often do you update streamfunction
    save_data = 0;                          %Save critical variable to vortex_data.mat?
    max_iter = 500;                         %If using gmres/bicgstab, max iterations

else
end

if demo_selection==3
    %Demonstration
    solution_method = 1;
    matrix_selection = 0;
    vorticity_type = 3;
    vorticity_selection = 0;
    fft_selection = 1;
    camera_options = [-37.5 -37.5 30 30];
    relres_selection = 0;
    wave_selection = 0;
    streamfunction_selection = 0;
    L = 20;
    m = 2^6;
    v = 0.001;
    num_relres_plots = 3;
    t_final = 60;
    psi_update = 1;
    save_data = 0;
else
end


if demo_selection==2
    %Obtain constants
    disp('Please enter computational domain.')
    disp('(recommended L = 20)')
    disp(' ')
    L = input('  L:  ');
    disp(' ')
```

```
disp('Please enter number of discretization points in x and y direction.')
disp('Must be power of 2^p (for p = 4, 5, 6, 7, 8)')
disp('(recommended m = 2^6)')
disp(' ')
m = input('  m:  ');
disp(' ')

while (m~=2^4) & (m~=2^5) & (m~=2^6) & (m~=2^7) & (m~=2^8)
    disp('Invalid ''m'', please enter a power of 2.')
    disp('(recommended m = 2^6)')
    disp(' ')
    m = input('  m:  ');
    disp(' ')
end

disp('Please enter diffusion coefficient.')
disp('(recommended v = 0.001)')
disp(' ')
v = input('  v:  ');
clc

%Obtain final time and psi_update
disp('How long do you want to run simulation?')
disp('(recommended t_final < 200)')
disp(' ')
t_final = input('  t_final:  ');
disp(' ')
disp('How often do you want to update the streamfunction?')
disp('(recommended psi_update = 1)')
disp(' ')
psi_update = input('  psi_update:  ');
clc


disp('Which elliptic solution method would you like to use?')
disp(' ')
disp('  1 = FFT (recommended)')
disp('  2 = A\b')
disp('  3 = LU Decomposition')
disp('  4 = BICGSTAB')
disp('  5 = GMRES')
disp('  6 = Spectral Implementation')
disp(' ')
solution_method = input('Please enter selection:  ');
disp(' ')
clc

switch solution_method
    case 1
        %FFT
        disp('Would you like to update streamfunction at every time step')
        disp('or only at discrete time steps?')
        disp(' ')
        disp('  1 = Every step (spectral method, recommended)')
        disp('  2 = Discrete points')
        disp(' ')
        fft_selection = input('Please enter selection:  ');
        disp(' ')
        clc

        if fft_selection==1
            disp('Would you like to rotate the camera during the movie?')
            disp(' ')
            disp('  1 = yes')
            disp('  2 = no')
            disp(' ')
            camera_selection = input('Please enter selection:  ');
            disp(' ')
            clc
        else
            disp('Would you like to look at the stream function?')
```

```
        disp(' ')
        disp('  1 = yes')
        disp('  2 = no')
        disp(' ')
        streamfunction_selection = input('Please enter selection:  ');
        clc
    end

    if camera_selection==1
        %Obtain user preferences on camera rotation
        disp('Please enter in camera rotation options.')
        disp(' ')
        az_start = input('  Starting azimuth (default is -37.5):  ');
        az_end = input('  Ending azimuth (default is -37.5):  ');
        disp(' ')
        el_start = input('  Starting elevation (default is 30):  ');
        el_end = input('  Ending elevation (default is 30):  ');
        clc
    else
        %Use a static camera with default values.
        az_start = -37.5;           %Default is -37.5
        az_end = -37.5;

        %Define elevation sweep for cool plotting
        el_start = 30;              %Default is 30
        el_end = 30;
    end

    disp('Would you like to look at wave wave numbers?')
    disp(' ')
    disp('  1 = yes')
    disp('  2 = no')
    disp(' ')
    wave_selection = input('Please enter selection:  ');
    disp(' ')
    clc


case 4
    %BICGSTAB
    disp('Would you like to view the plots for BICGSTAB and GMRES?')
    disp(' ')
    disp('  1 = yes')
    disp('  2 = no')
    disp(' ')
    relres_selection = input('Please enter selection:  ');
    disp(' ')
    clc

    if relres_selection==1
        disp('How many plots of relative residual would you like to see?')
        disp(' ')
        num_relres_plots = input('Please enter selection:  ')
        clc
    else
    end

case 5
    %GMRES
    disp('Would you like to view the plots for BICGSTAB and GMRES?')
    disp(' ')
    disp('  1 = yes')
    disp('  2 = no')
    disp(' ')
    relres_selection = input('Please enter selection:  ');
    disp(' ')
    clc

    if relres_selection==1
        disp('How many plots of relative residual would you like to see?')
        disp(' ')
```

```
                num_relres_plots = input('Please enter selection:  ')
                clc
            else
            end

        otherwise

    end


    disp('Would you like to view the A, B, and C matrices?')
    disp(' ')
    disp('  1 = Yes')
    disp('  2 = No')
    disp(' ')
    matrix_selection = input('Please enter selection:  ');
    disp(' ')
    clc

    disp('What type of initial vorticity would you like?')
    disp(' ')
    disp('  1 = Simple Gaussian')
    disp('  2 = Two oppositely "charged" Gaussians')
    disp('  3 = Two same "charged" Gaussians')
    disp('  4 = Two pairs of oppositely "charged" vortices which collide with each
other')
    disp('  5 = Random assortment of vorticies')
    disp('  6 = Two same "charged" Gaussians, one with half the magnitude')
    disp('  7 = Two oppositely "charged" Gaussians, one with half the magnitude')
    disp('  8 = Two pairs of oppositely "charged vortices, one which catches up with the
other')
    disp(' ')
    vorticity_type = input('Please enter selection:  ');
    clc

    disp('Would you like to view initial vorticity?')
    disp(' ')
    disp('  1 = Yes')
    disp('  2 = No')
    disp(' ')
    vorticity_selection = input('Please enter selection:  ');
    clc

    disp('Would you like to save critical variables? (Data saved to vortex_data.mat)')
    disp(' ')
    disp('  1 = Yes')
    disp('  2 = No')
    disp(' ')
    save_data = input('Please enter selection:  ');
    clc


end
```

create_matrices.m

```
function [A,B,C] = create_matrices(n,m,delta,matrix_selection)

%Create the A, B, and C matrices
%
%INPUT:     -number of discretizations (n)
%           -total size of matrix (m)
%           -spatial step size (delta)
%           -view sparse matrices or not (matrix_selection)
%
%OUTPUT:    -A matrix
%           -B matrix
%           -C matrix
```

```
e0 = zeros(n,1);
e1 = ones(n,1);

e2 = e1;
e4 = e0;

%We want to modify e2 and e4.  e2 needs to have 0's once in a while and e4
%needs to have 1's once in a while
for j=1:m
    e2(m*j) = 0;    %Place a zero every m th element.
    e4(m*j) = 1;    %Place a 1 every m th element.
end

%Now create vectors which will be placed below the main diagonals.  These
%are similar to the ones above the diagonal, but shifted.
e3(2:n,1) = e2(1:n-1,1);
e3(1,1) = e2(n,1);

e5(2:n,1) = e4(1:n-1,1);
e5(1,1) = e4(n,1);

%Create A matrix
A = spdiags([e1    e1  e5  e2  -4*e1  e3  e4  e1  e1],...
          [-(n-m) -m -m+1  -1    0    1   m-1  m (n-m)],...
          n,n);
A = A*(1/delta^2);

%Create a non-singular version of this matrix to be used by the elliptic
%solve.
A_elliptic = A;
A_elliptic(1,1) = 0;



%-------------------------Create B Matrix---------------------------
B = spdiags([e1  -1*e1  e1  -1*e1],...
          [-(n-m) -m    m    (n-m)],...
          n,n);

B = B*(1/(2*delta));


%-------------------------Create C Matrix---------------------------
C = spdiags([e5  -1*e2    e3  -1*e4],...
            [-m+1   -1     1    m-1],...
             n,n);

C = C*(1/(2*delta));

%Do we want to see the matrices?
if matrix_selection==1
    figure
    spy(A)

    figure
    spy(B)

    figure
    spy(C)
else
end
```

create_vorticity.m

```
function [X,Y,wo] = create_vorticity(xyspan,vorticity_type,vorticity_selection,L)

%Create the initial vorticity for the problem.
```

```
%
%INPUT:     -x and y discretization (xyspan)
%           -desired type of vorticity (vorticity_type)
%           -plot vorticity or not? (vorticity_selection)
%           -computational domain (L)
%
%OUTPUT:    -X (meshgrid x coordinates)
%           -Y (meshgrid y coordinates)
%           -wo (initial vorticity corresponding to X and Y)

[X,Y] = meshgrid(xyspan,xyspan);

switch vorticity_type
    case 1
        %Simple gaussian
        wo = exp(-2*X.^2 - ((Y.^2)./L));

    case 2
        %Two opposite gaussians
        %We want to shift the gaussian to one side.  Move one of them over
        %by 20% of the computational domain
        offset = 0.2*(L/2);
        positive = exp(-(X + offset).^2 - (Y.^2)./1);
        negative = -exp(-(X - offset).^2 - (Y.^2)./1);

        wo = positive + negative;

    case 3
        %Two same gaussians
        wo = Y.^2.*exp(-X.^2.-Y.^2);

        offset = 0.2*(L/2);
        positive = exp(-(X + offset).^2 - (Y.^2)./1);
        negative = exp(-(X - offset).^2 - (Y.^2)./1);

        wo = positive + negative;

    case 4
        %Two opposite gaussian pairs which collide
        %Let's use two pairs of opposite gaussians.  One pair will be
        %symmetric about 20% of the centerline and shifted in y by 40% of
        %the computational domain.  The other set will be
        x_offset = 0.2*(L/2);
        y_offset = 0.4*(L/2);

        positive1 = exp(-(X + x_offset).^2 - (Y + y_offset).^2);
        negative1 = -exp(-(X - x_offset).^2 - (Y + y_offset).^2);

        positive2 = -exp(-(X + x_offset).^2 - (Y - y_offset).^2);
        negative2 = exp(-(X - x_offset).^2 - (Y - y_offset).^2);

        wo = positive1 + negative1 + positive2 + negative2;

    case 5
        %Random assortment

        %How many random vorticies are desired?
        num_vorticies = 20;

        %Initialize wo matrix
        [m,m] = size(X);
        wo = zeros(m,m);

        for counter=1:num_vorticies

            %let's make the magnitude vary from -1 to 1
            magnitude = 2*(rand - 0.5);

            %We want to only create gaussians that are within the domain.
            %Let's restrict the offsets to only be 75% of the domain.
            x_offset = 1.5*(rand - 0.5)*(L/2);
```

```
            y_offset = 1.5*(rand - 0.5)*(L/2);

            wo = wo + 2*(rand - 0.5)*exp(-(X + x_offset).^2 - (Y + y_offset).^2);

        end

    case 6
        %Two same gaussians next to each other with different magnitudes
        wo = Y.^2.*exp(-X.^2.-Y.^2);

        offset = 0.2*(L/2);
        tall_one = exp(-(X + offset).^2 - (Y.^2)./1);
        short_one = 0.5*(exp(-(X - offset).^2 - (Y.^2)./1));

        wo = tall_one + short_one;

    case 7
        %Two opposite gaussiants with different magnitudes
        offset = 0.2*(L/2);
        positive = 0.5*(exp(-(X + offset).^2 - (Y.^2)./1));
        negative = -exp(-(X - offset).^2 - (Y.^2)./1);

        wo = positive + negative;

    case 8
        %Two pairs of gaussians.  Both traveling same direction.  One
        %catches up with the other.
        x_offset = 0.2*(L/2);
        y_offset = 0.3*(L/2);

        positive1 = -0.5*(exp(-(X + x_offset).^2 - (Y + y_offset).^2));
        negative1 = 0.5*exp(-(X - x_offset).^2 - (Y + y_offset).^2);

        positive2 = -(exp(-(X + x_offset).^2 - (Y - y_offset).^2));
        negative2 = (exp(-(X - x_offset).^2 - (Y - y_offset).^2));

        wo = positive1 + negative1 + positive2 + negative2;



    otherwise
        %Assume a simple gaussian
        wo = exp(-2*X.^2 - ((Y.^2)./20));
end

if vorticity_selection==1
    figure
    surf(X,Y,wo)
    shading interp
    lightangle(-37.5,30)
    lighting flat
    xlabel('x')
    ylabel('y')
    switch vorticity_type
        case 1
            title('Initial Vorticity \omega for Stretched Gaussian')
        case 2
            title('Initial Vorticity \omega for 2 Oppositely Charged Gaussians')
        case 3
            title('Initial Vorticity \omega for 2 Same Charged Gaussians')
        case 4
            title('Initial Vorticity \omega for 2 Pair of Colliding Gaussians')
        case 5
            title('Initial Vorticity \omega for Random Vortices')
        case 6
            title('Initial Vorticity \omega for Two Same Gaussians with Different
Magnitude')
        case 7
            title('Initial Vorticity \omega for Two Opposite Gaussians with Different
Magnitude')
        case 8
```

```
            title('Initial Vorticity \omega for Two Pairs of Colliding Vortices')
        otherwise
    end
    drawnow
else
end
```

advection_fft.m

```
function dw_dt_col = advection_fft(t,w_col,dummy,A,B,C,v,KX,KY,m,t_final)

global trigger
global ode_t
global process_time

%Solve the advection-diffusion equations to advance the vorticity into the
%future.  Solve the elliptic solve at every time step using FFT to update
%the streamfunction at every step.
%
%INPUTS:    -time (t)
%           -initial vorticity (w)
%           -dummy variable for the options
%           -A matrix which performs del^2 (A)
%           -B matrix which performs d/dx (B)
%           -C matrix which performs d/dy (C)
%           -diffusion coefficient (v)
%           -KX
%           -KY
%           -m
%           -final time (t_final)

n = m^2;

%Reshape the vorticity into a matrix
w = reshape(w_col,m,m);

%Calculate the FFT of the vorticity
w_t = fft2(w);

%Solve del^2*psi = w in fourier domain
psi_t = (-w_t)./(KX.^2 + KY.^2);

%Shift psi back into spatial domain
psi = ifft2(psi_t);

%We need to reshape the stream function into a column
psi_col = reshape(psi,n,1);

%Solve ode
dw_dt_col = (C*psi_col).*(B*w_col) - (B*psi_col).*(C*w_col) + (v*A*w_col);



%How many percent done are we?
percent = t/t_final;
percent_10 = floor(percent*10);


switch percent_10
    case 1
        if trigger(1)~=1
            disp(' ')
            disp('    Solution Process')
            disp(['    10% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(2) = cputime - ode_t;
            trigger(1) = 1;
        else
        end
```

```
    case 2
        if trigger(2)~=1
            disp(['    20% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(3) = cputime - ode_t;
            trigger(2) = 1;
        else
        end

    case 3
        if trigger(3)~=1
            disp(['    30% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(4) = cputime - ode_t;
            trigger(3) = 1;
        else
        end

    case 4
        if trigger(4)~=1
            disp(['    40% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(5) = cputime - ode_t;
            trigger(4) = 1;
        else
        end

    case 5
        if trigger(5)~=1
            disp(['    50% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(6) = cputime - ode_t;
            trigger(5) = 1;
        else
        end

    case 6
        if trigger(6)~=1
            disp(['    60% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(7) = cputime - ode_t;
            trigger(6) = 1;
        else
        end

    case 7
        if trigger(7)~=1
            disp(['    70% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(8) = cputime - ode_t;
            trigger(7) = 1;
        else
        end

    case 8
        if trigger(8)~=1
            disp(['    80% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(9) = cputime - ode_t;
            trigger(8) = 1;
        else
        end

    case 9
        if trigger(9)~=1
            disp(['    90% Complete.    ',num2str(cputime - ode_t),' sec elapsed'])
            process_time(10) = cputime - ode_t;
            trigger(9) = 1;
        else
        end

    otherwise
end
```

analyze_wave_function.m

```
function analyze_wave_function(xyspan,kx)

%Creates plots of random wave numbers
%
%INPUT:     -xyspan
%           -wave numbers (kx)
%
%OUTPUT:    -Plots of cos and sin functions with corresponding wave numbers

%Which wave numbers should we look at?
no1 = 1;
no2 = 2;
no3 = 3;

figure
subplot(3,1,1)
plot(xyspan,cos(kx(no1)*xyspan))
axis([min(xyspan) max(xyspan) -1 1.1])
title('Random cos(k_xx) Functions')
grid
legend(['k_x = ',num2str(kx(no1))])

subplot(3,1,2)
plot(xyspan,cos(kx(no2)*xyspan))
axis([min(xyspan) max(xyspan) -1 1])
grid
legend(['k_x = ',num2str(kx(no2))])

subplot(3,1,3)
plot(xyspan,cos(kx(no3)*xyspan))
axis([min(xyspan) max(xyspan) -1 1])
grid
legend(['k_x = ',num2str(kx(no3))])
xlabel('x')

figure
subplot(3,1,1)
plot(xyspan,sin(kx(no1)*xyspan))
axis([min(xyspan) max(xyspan) -1 1])
title('Random sin(k_xx) Functions')
grid
legend(['k_x = ',num2str(kx(no1))])

subplot(3,1,2)
plot(xyspan,sin(kx(no2)*xyspan))
axis([min(xyspan) max(xyspan) -1 1])
grid
legend(['k_x = ',num2str(kx(no2))])

subplot(3,1,3)
plot(xyspan,sin(kx(no3)*xyspan))
axis([min(xyspan) max(xyspan) -1 1])
grid
legend(['k_x = ',num2str(kx(no3))])

figure
plot([1:length(xyspan)],kx,'rx')
axis([0 length(xyspan) min(kx) max(kx)])
title('Eligible Wave Numbers')
xlabel('Vector Position')
ylabel('Wave Number')
grid
```

ask_question.m

```
function ask_question

%Asks a riddle to ponder while we're waiting for the fft to perform the
%elliptic solve.

riddle_no = floor(rand*10);

disp(' ')
switch riddle_no
    case 0
        disp('------------------------------------------------------------------')
        disp('How much wood could a wood chuck chuck if a wood chuck could')
        disp('chuck wood?')
        disp('------------------------------------------------------------------')

    case 1
        disp('------------------------------------------------------------------')
        disp('3 guys walk into a model and ask for a room.  The managers says')
        disp('"That''s $30 for a room."  Each guy takes out a ten dollar bill')
        disp('and gives it to the manager then walks up to their room.  Later,')
        disp('the manager realizes that he overchaged them, it was only $25,')
        disp('for a room.')
        disp(' ')
        disp('He gives the bellboy a $5 bill and tells him to return it to the')
        disp('three guys.  On the way up the elevator, the bellboy thinks,')
        disp('"3 guys can''t split a $5 bill between themselves." so he pockets')
        disp('the $5 bill and instead, gives the guys 3 one dollar bills.')
        disp(' ')
        disp('This means that each guy paid $9.  $9 per guy times 3 guys equals')
        disp('$27 dollars.  That plus the $2 that the bellboy has equals only')
        disp('$29 dollars.  Where did the extra dollar go?')
        disp('------------------------------------------------------------------')

    case 2
        disp('------------------------------------------------------------------')
        disp('An archeologist is performing a dig in ancient Persia when he')
        disp('discovers an extremely old coin which is dated 500 BC.  Overjoyed,')
        disp('he takes the coin back to a collector to see how much it is worth.')
        disp('The collector takes one look at it and says, "There''s no way')
        disp('this coin is that old."  How did he know?')
        disp('------------------------------------------------------------------')

    case 3
        disp('------------------------------------------------------------------')
        disp('A Swiss plane carrying 5 Mexican convicts and 5 American convicts')
        disp('crashes exactly on the US/Mexico border.  Where do you bury the')
        disp('survivors?')
        disp('------------------------------------------------------------------')

    case 4
        disp('------------------------------------------------------------------')
        disp('I can prove that 2 + 2 = 2')
        disp(' ')
        disp('    Let a = 2')
        disp('        x = 2')
        disp(' ')
        disp('                x = a')
        disp('              x^2 = ax')
        disp('         x^2 - a^2 = ax - a^2')
        disp('    (x - a)(x + a) = a(x - a)')
        disp('             x + a = a')
        disp('             2 + 2 = 2')
        disp(' ')
        disp('Where did I go wrong?')
        disp('------------------------------------------------------------------')


    otherwise
        disp('------------------------------------------------------------------')
        disp('3 guys walk into a model and ask for a room.  The managers says')
        disp('"That''s $30 for a room.  Each guy takes out a ten dollar bill')
```

```
        disp('and gives it to the manager then walks up to their room.  Later,')
        disp('the manager realizes that he overchaged them, it was only $25,')
        disp('for a room.')
        disp(' ')
        disp('He gives the bellboy a $5 bill and tells him to return it to the')
        disp('three guys.  On the way up the elevator, the bellboy thinks,')
        disp('"3 guys can''t split a $5 bill between themselves." so he pockets')
        disp('the $5 bill and instead, gives the guys 3 one dollar bills.')
        disp(' ')
        disp('This means that each guy paid $9.  $9 per guy times 3 guys equals')
        disp('$27 dollars.  That plus the $2 that the bellboy has equals only')
        disp('$29 dollars.  Where did the extra dollar go?')
        disp('-----------------------------------------------------------------')
end
```

display_parameters.m

```
function display_parameters(L,m,v,vorticity_type,t_final,psi_update,save_data)

%Display the parameters being used
%
%INPUTS:    -None
%
%OUTPUTS:   -Display parameters to the screen

disp(' ')
disp('    Solution Parameters Being Applied')
disp('    ------------------------------------------------')
disp(['    L:                  ',num2str(L)])
disp(['    m:                  ',num2str(m)])
disp(['    v:                  ',num2str(v)])
disp(['    t_final:            ',num2str(t_final),' seconds'])
disp(['    psi_update:         ',num2str(psi_update),' seconds'])

switch vorticity_type
    case 1
        disp('    Initial Vorticity:  Stretched Gaussian')
    case 2
        disp('    Initial Vorticity:  2 Opposite Gaussians')
    case 3
        disp('    Initial Vorticity:  2 Same Gaussians')
    case 4
        disp('    Initial Vorticity:  2 Pairs of Colliding Gaussians')
    case 5
        disp('    Initial Vorticity:  Random Vortices')
    case 6
        disp('    Initial Vorticity:  2 Same Gaussians w/ Different Magnitudes')
    case 7
        disp('    Initial Vorticity:  2 Opposite Gaussians w/ Different Magnitudes')
    case 8
        disp('    Initial Vorticity:  2 Pairs of Guassians (one catches up w/ other
pair)')
    otherwise
        disp('    Initial Vorticity:  Stretched Gaussian')
end

if save_data==1
    disp('    Save data?:         Yes (vortex_data.mat)')
else
    disp('    Save data?:         No')
end
disp('    ------------------------------------------------')

disp('    Now solving, please wait...')
```

view_movie.m

```
%Play movie of data and also create plots of vorticity coefficient and
%process time for the previous run.  This is designed to be run
%independently an only analyze data, not create it.

clc
close all

%-------------------User Preferences---------------------------------
type = 3;       %1 = surf
                %2 = pcolor (with streamfunction if appilcable)
                %3 = both surf and pcolor
                %4 = vorticity coefficient and process time plots

%Do we need to look load old data?
load_data = 2;  %1 = yes
                %2 = no

%Do you want to change the view?
%Define the azimuth sweep for cool plotting
az_start = -37.5;           %Default is -37.5
az_end = 37.5;

%Define elevation sweep for cool plotting
el_start = 10;              %Default is 30
el_end = 30;

%What about the lighting angle, it should be constant
az_light = -37.5;
el_light = 30;
%--------------------------------------------------------------------


%Set axis limits
xmin = -L/2;
xmax = L/2;
ymin = -L/2;
ymax = L/2;
zmin = -1;
zmax = 1;


%Do we need to load old data?
if load_data==1
    load vortex_data
else
end

%Are we dealing with the streamfunction as well?
[rows,cols] = size(psio_col_data);

if rows > 1
    %Streamfunction data is stored
    psi_col = real(psio_col_data);
    w_col =  wo_col_data;
    t = t_data;
else
end

%create an array for camera rotation.
az_array = linspace(az_start,az_end,length(t));

el_array = linspace(el_start,el_end,length(t));

switch type
    case 1
        %---------------------------------surf-----------------------------------
        figure
        for counter=1:length(t)
```

```
        w = reshape(w_col(counter,:),m,m);
        surf(X,Y,w)
        shading interp
        axis([xmin xmax ymin ymax zmin zmax])
        title(['Time = ',num2str(round(t(counter)))])

        %What is current view?
        view(az_array(counter),el_array(counter));

        %Set the lighting angle
        lightangle(az_light,el_light)

        M(counter) = getframe;

        drawnow
    end

case 2
    %----------------------------pcolor-------------------------------
    figure
    for counter=1:length(t)
        w = reshape(w_col(counter,:),m,m);
        pcolor(X,Y,w)
        shading interp
        title(['Time = ',num2str(round(t(counter)))])
        colorbar
        drawnow
    end

    %Let's see if we are plotting the steamfunction as well
    if rows > 1
        counter = 1;
        for counter=1:length(t)
            psi = reshape(psi_col(counter,:),m,m);
            pcolor(X,Y,psi)
            shading interp
            title(['Time = ',num2str(round(t(counter)))])
            colorbar
            drawnow

            M(counter) = getframe;
        end
    else
    end

case 3
    %------------------------surf and pcolor--------------------------
    figure
    disp('-----------------------------------------------------')
    disp('- Enlarge Figure then press any key to continue... -')
    disp('-  (movie will play 2 seconds after pressing key)  -')
    disp('-----------------------------------------------------')
    pause
    pause(2)
    for counter=1:length(t)
        w = reshape(w_col(counter,:),m,m);

        subplot(1,2,1)
        surf(X,Y,w)
        title(['Azimuth = ',num2str(round(az_array(counter))),'.  Elevation = ',...
               num2str(round(el_array(counter)))])
        view(az_array(counter),el_array(counter));
        axis([xmin xmax ymin ymax zmin zmax])
        shading interp

        %Set the lighting angle
        lightangle(az_light,el_light)

        subplot(1,2,2)
        pcolor(X,Y,w-1)
        shading interp
```

```
            colorbar
            title(['Time = ',num2str(round(t(counter)))])

            %M(counter) = getframe;
            drawnow
        end

    case 4
        %----------Vorticity coefficient and process time--------------
        for counter=1:length(t)
            w = reshape(w_col(counter,:),m,m);

            [C_w,C_w2] = vorticity_coefficient(w,m,m);
            C_w_data(counter) = C_w;
            C_w2_data(counter) = C_w2;
        end

        figure
        plot(t,C_w_data)
        axis([0 max(t) min(C_w_data) max(C_w_data)])
        legend(['\nu = ',num2str(v)])
        title('Vorticity Coefficient vs. Time')
        xlabel('Time (sec)')
        ylabel('Vorticity Coefficient')
        grid

        figure
        percent = [0 10 20 30 40 50 60 70 80 90 100];
        plot(percent,process_time)
        grid
        title('Total Time Elapsed vs. Percentage Complete')
        legend(['\nu = ',num2str(v)],2)
        xlabel('Percentage Complete (%)')
        ylabel('Total Time Elapsed (sec)')

    otherwise
end
```

view_subplots.m

```
%Create subplots for report.  Breaks into 9 subplots.  This is designed to
%be run independently and only analyze data, not create it.

clc
close all

%-------------------User Preferences---------------------------------
type = 1;       %1 = pcolor
                %2 = surf

%What is the last time value that you want?
t_final = 300;


%What about the lighting angle, it should be constant
az_light = -37.5;
el_light = 30;
%-------------------User Preferences---------------------------------

%Check to see if the psio_col_data has more than 1 row.
[rows,cols] = size(psio_col_data);

if rows > 1
    %Streamfunction data is stored
    psi_col = real(psio_col_data);
    w_col =  wo_col_data;
    t = t_data;
else
end
```

```
last_index = max(find(t<t_final));

%How many data "points" are there?
num_points = length(t(1:last_index));

index = round(num_points/9);


%---------------------------Vorticity--------------------------------------

figure
if type==1
    %Pcolor plots
    subplot(3,3,1)
    w = reshape(w_col(1,:),m,m);
    pcolor(X,Y,w)
    shading interp
    title(['Time = ',num2str(t(1))])

    for counter=2:9
        subplot(3,3,counter)
        w = reshape(w_col((counter-1)*index,:),m,m);
        pcolor(X,Y,w)
        shading interp
        title(['Time = ',num2str(t((counter-1)*index))])
    end

else
    %Surf Plots

    %Figure out what is the starting axis to hold (ie what is the maximum z
    %value at t = 0?)
    zmax = max(w_col(1,:));
    zmin = min(w_col(1,:));

    axis_settings = [min(min(X)) max(max(X)) min(min(Y)) max(max(Y)) zmin zmax];

    subplot(3,3,1)
    w = reshape(w_col(1,:),m,m);
    surf(X,Y,w)
    axis(axis_settings);
    shading interp
    lightangle(az_light,el_light)
    title(['Time = ',num2str(t(1))])

    for counter=2:9
        subplot(3,3,counter)
        w = reshape(w_col((counter-1)*index,:),m,m);
        surf(X,Y,w)
        axis(axis_settings);
        shading interp
        lightangle(az_light,el_light)
        title(['Time = ',num2str(t((counter-1)*index))])
    end

end

%---------------------------Streamfunction---------------------------------------
if rows > 1
    figure
    subplot(3,3,1)
    psi = reshape(psi_col(1,:),m,m);
    pcolor(X,Y,psi)
    shading interp
    title(['Time = ',num2str(t(1))])

    for counter=2:9
        subplot(3,3,counter)
        psi = reshape(psi_col((counter-1)*index,:),m,m);
        pcolor(X,Y,psi)
```

```
        shading interp
        title(['Time = ',num2str(t((counter-1)*index))])
    end

else
end
```

vorticity_coefficient.m

```
function [C_w,C_w2] = vorticity_coefficient(w,ML,NL)

%Calculate the vorticity coefficient for the computational domain.
%
%INPUT:     -vorticity over computational domain in matrix form (w)
%           -number of points to sample in y direction (ML)
%           -number of points to sample in x direction (NL)
%
%OUTPUT:    -vorticity coefficient (C_w)

%Initialize C_w
w_total = 0;

%Round number of discretization points
ML = floor(ML);
NL = floor(NL);

[rows,cols] = size(w);

%how many do we jump by in the x and y interval?
y_interval = floor(rows/NL);
x_interval = floor(rows/ML);


for n=1:ML
    for m=1:NL

        %What is the current column and row?
        current_col = n*x_interval;
        current_row = m*y_interval;

        w_total = w_total + abs(w(current_row,current_col));
    end
end

C_w = w_total/(ML*NL);

C_w2 = sum(sum(abs(w)))/(rows*cols);
```

Appendix C: Derivation of Equations

Equations of Motion

The derivation of the equations of motion relies on the control volume shown below in Figure 43.

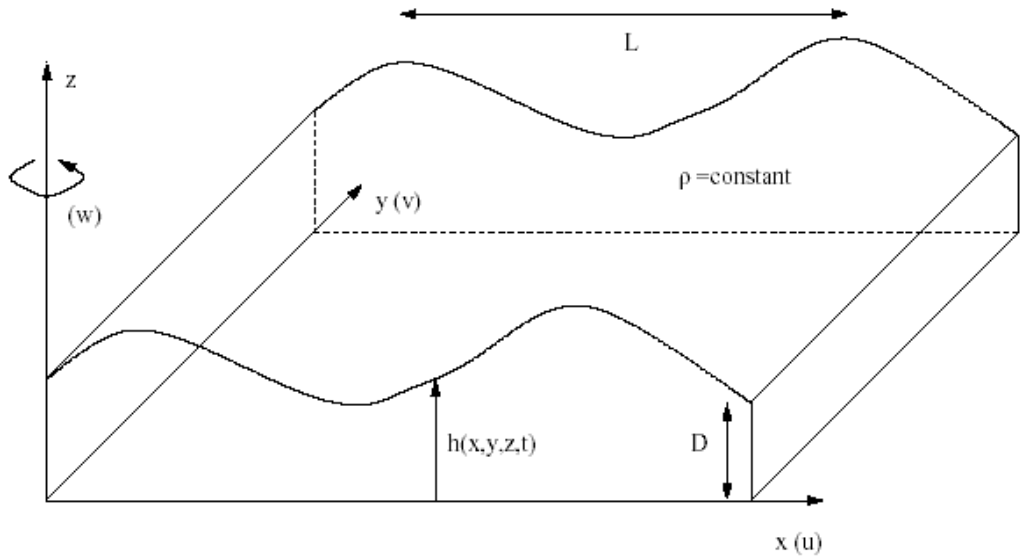


Figure 43: Control volume used to derive equations of motion

For the shallow water approximation to be valid, we require that the ratio of D/L be small. The vorticity of the fluid in the z direction is of primary interest. This is defined as shown in Equation 21.

$$\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$$

Equation 21

The total mass of the control volume can be described by Equation 22.

$$M = \int_{y_1}^{y_2} \int_{x_1}^{x_2} \rho(x,y) h(x,y,t)\, dxdy$$

Equation 22

The total mass must be conserved and the conservation of mass equation can be written as (assuming that density is constant)

$$\frac{d}{dt}\int_{y_1}^{y_2}\int_{x_1}^{x_2} h\,dxdy+\int_{y_1}^{y_2}\int_{x_1}^{x_2}\frac{d}{dx}(uh)dxdy+\int_{y_1}^{y_2}\int_{x_1}^{x_2}\frac{d}{dy}(vh)\,dxdy=0$$

Equation 23

Since all of these double integrations are over the same surface, this obviously implies that the integrand must be zero. Assuming that the height is constant, the incompressible version of the conservation of mass equation is given by Equation 24 shown below.

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial x}=0$$

Equation 24

A similar analysis with the same assumptions can be performed with the momentum equation to yield the following two equations

$$\frac{\partial u}{\partial t}+2u\frac{\partial u}{\partial x}+\frac{\partial(uv)}{\partial y}=fv$$

Equation 25

$$\frac{\partial v}{\partial t}+2v\frac{\partial v}{\partial y}+\frac{\partial(uv)}{\partial x}=-fu$$

Equation 26

Taking the y derivative of Equation 25, the x derivative of Equation 26, subtracting the first equation from the second, and by recalling the definition of vorticity and using Equation 24, the governing equation of motion for the fluid in the absence of friction is given by Equation 27.

$$\frac{\partial \omega}{\partial t}+u\frac{\partial \omega}{\partial x}+v\frac{\partial \omega}{\partial y}=0$$

Equation 27

The problem becomes easier to work with by defining what is known as the streamfunction. This is defined in terms of derivatives and velocity.

$$u = -\frac{\partial \psi}{\partial y} \qquad v = \frac{\partial \psi}{\partial x}$$

Equation 28

At this point, it becomes prudent to introduce the Laplacian operator. It is defined as

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

Equation 29

The streamfunction is related to the vorticity through the Laplacian operator.

$$\omega = \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \nabla^2 \psi$$

Equation 30

Equation 30 is known as the elliptic equation.

In the presence of friction, the governing equations reduce to

$$\frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega + \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y}$$

Equation 31

Equation 31 is known as the advection equation.

The problem can now be solved by using Equation 30 to solve for the streamfunction and then using the streamfunction and vorticity in Equation 31 to advance the vorticity into the future. Once a new value of vorticity is found, Equation 30 can be used again to find a new streamfunction and the process repeats itself.

Finite Difference Discretization

Solving this problem numerically requires a discretization of both Equation 30 and Equation 31. Recall that the second order discretization of a second derivative of a function is given by

$$\frac{d^2 f(t)}{dt^2} = \frac{f(t+\Delta t) - 2f(t) + f(t-\Delta t)}{\Delta t^2}$$

Equation 32

Therefore, starting with the elliptic equation, Equation 30 can be discretized using a second order accurate scheme for the second derivative in both x and y directions.

$$\omega(x,y,t) = \frac{\psi(x+\Delta x,y,t) - 2\psi(x,y,t) + \psi(x-\Delta x,y,t)}{\Delta x^2} + \frac{\psi(x,y+\Delta y,t) - 2\psi(x,y,t) + \psi(x,y-\Delta y,t)}{\Delta y^2}$$

Equation 33

For convenience, we'll choose $\Delta x = \Delta y = \delta$ and for a given t, denote

$$\psi_{mn} = \psi(m,n,t_o)$$

With these simplifications, Equation 33 reduces to

$$\frac{-4\psi_{m,n} + \psi_{m-1,n} + \psi_{m+1,n} + \psi_{m,n+1} + \psi_{m,n-1}}{\delta^2} = \omega_{m,n}$$

Equation 34

From Equation 34 it can be seen that the vorticity at any point in the discretization depends on the stream function at 4 neighboring points. Furthermore, with the periodic boundary conditions

$$\psi_{N+1,n} = \psi_{1,n}$$
$$\psi_{m,N+1} = \psi_{m,1}$$

A similar procedure can be performed for discretizing the partial derivative of a function in the x direction and the partial derivative of a function in the y direction. This is shown below (recall that $\Delta x = \Delta y = \delta$).

$$\frac{\partial \psi_{m,n}}{\partial x} = \frac{\psi_{m+1,n} - \psi_{m-1,n}}{2\delta}$$

Equation 35

$$\frac{\partial \psi_{m,n}}{\partial y} = \frac{\psi_{m,n+1} - \psi_{m,n-1}}{2\delta}$$

Equation 36

A simple stencil of this pattern for N = 4 is shown below in Figure 44.

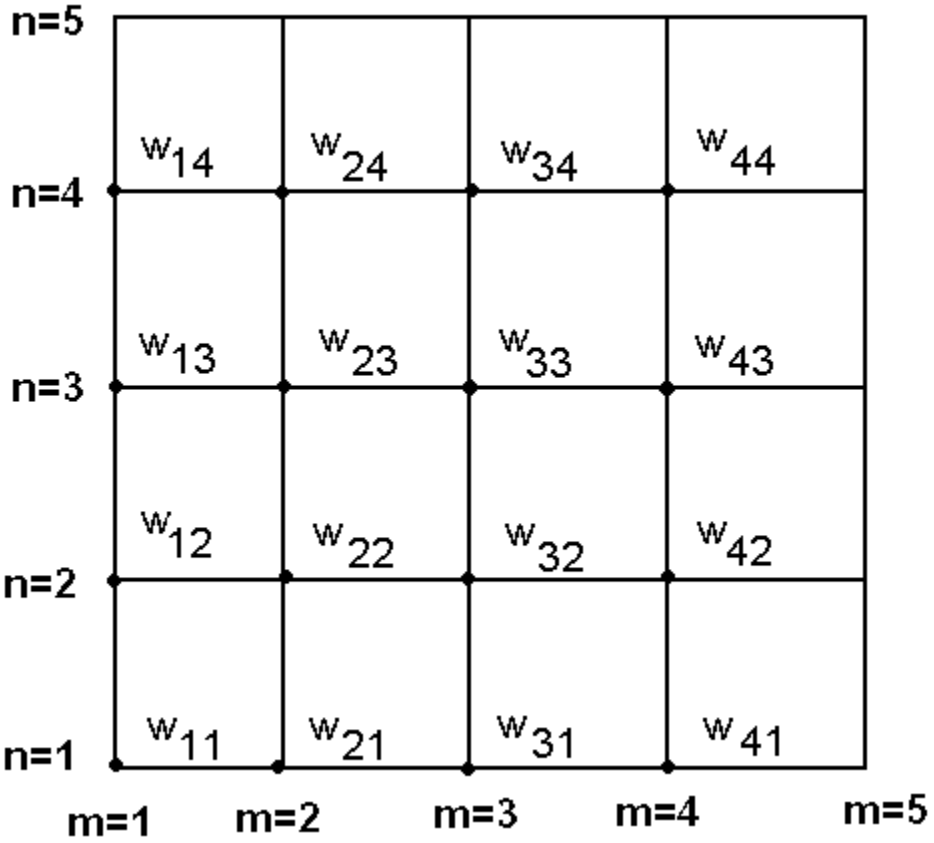


Figure 44: Pattern for vorticity and streamfunction at discrete points

Since the program makes extensive use of the MESHGRID and RESHAPE commands, it is important that the mechanics of these functions are understood. First, two vectors must be defined which represent the discretization in the x and y directions

$$x = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix}$$
$$y = \begin{pmatrix} y_1 & y_2 & y_3 & y_4 \end{pmatrix}$$

The MESHGRID command is then used to create two matrices which represent these vectors in two dimensions as shown below.

[X,Y] = meshgrid(x,y)

$$X = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \end{pmatrix} \qquad Y = \begin{pmatrix} y_1 & y_1 & y_1 & y_1 \\ y_2 & y_2 & y_2 & y_2 \\ y_3 & y_3 & y_3 & y_3 \\ y_4 & y_4 & y_4 & y_4 \end{pmatrix}$$

The X and Y matrices can then be used to generate an initial vorticity.

$$\omega_o = f\left(X, Y\right)$$

The important thing to notice is that the resulting matrix, ω_o, will have values of vorticity at the different points stored in the following fashion.

$$\omega_o = \begin{pmatrix} \omega_{11} & \omega_{21} & \omega_{31} & \omega_{41} \\ \omega_{12} & \omega_{22} & \omega_{32} & \omega_{42} \\ \omega_{13} & \omega_{23} & \omega_{33} & \omega_{43} \\ \omega_{14} & \omega_{24} & \omega_{34} & \omega_{44} \end{pmatrix}$$

By comparing this with the stencil shown in Figure 44, one can realize that MatLab stores values of vorticity which are reflected over the horizontal middle line. That is, the value of vorticity in the upper left corner of the matrix ω_o is actually the value of vorticity in the bottom left corner of the stencil. Similarly, the value of vorticity in the upper right corner of the matrix ω_o is actually the value of vorticity in the bottom right corner of the stencil. This is important because when the reshape command is used to turn this matrix into a vector, the resulting column vector will be of the form.

wo_col = reshape(wo,16,1)

$$\omega_{o\,col} = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \omega_{14} & \omega_{21} & \omega_{22} & \omega_{23} & \omega_{24} & \omega_{31} & \omega_{32} & \omega_{33} & \omega_{34} & \omega_{41} & \omega_{42} & \omega_{43} & \omega_{44} \end{pmatrix}^T$$

Therefore, when creating the discretization of the Laplacian operator (using Equation 34) and the partial derivatives (using Equation 35 and Equation 36), the matrix must be constructed such that it is compliant with this ordering of points. Equation 34 written in matrix form with this ordering is shown below as

$$\frac{1}{\delta^2}\begin{pmatrix}
-4 & 1 & & 1 & 1 & & & & & & & 1 & & & & \\
1 & -4 & 1 & & & 1 & & & & & & & 1 & & & \\
 & 1 & -4 & 1 & & & 1 & & & & & & & 1 & & \\
1 & & 1 & -4 & 1 & & & 1 & & & & & & & & 1 \\
1 & & & 1 & -4 & 1 & & 1 & 1 & & & & & & & \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & & & & & \\
 & & & 1 & 1 & & 1 & -4 & 1 & & & 1 & & & & \\
 & & & & 1 & & & 1 & -4 & 1 & & 1 & 1 & & & \\
 & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & \\
 & & & & & & 1 & & & 1 & -4 & 1 & & & 1 & \\
 & & & & & & & 1 & 1 & & 1 & -4 & 1 & & & 1 \\
1 & & & & & & & & 1 & & & 1 & -4 & 1 & & 1 \\
 & 1 & & & & & & & & 1 & & & 1 & -4 & 1 & \\
 & & 1 & & & & & & & & 1 & & & 1 & -4 & 1 \\
 & & & 1 & & & & & & & & 1 & 1 & & 1 & -4
\end{pmatrix}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix} = \begin{pmatrix}\omega_{11}\\ \omega_{12}\\ \omega_{13}\\ \omega_{14}\\ \omega_{21}\\ \omega_{22}\\ \omega_{23}\\ \omega_{24}\\ \omega_{31}\\ \omega_{32}\\ \omega_{33}\\ \omega_{34}\\ \omega_{41}\\ \omega_{42}\\ \omega_{43}\\ \omega_{44}\end{pmatrix}$$

Equation 37

As can be seen, this is a sparse matrix with only 9 diagonals that result in anything other than a zero entry. The large, sparse matrix can be referred to as the A matrix, which performs the Laplacian operation on the vector representing the streamfunction. This now becomes a large matrix equation of the form $A\overline{\psi} = \overline{\omega}$. That is, given a vorticity in column form, one can use a multitude of method to solve this for the streamfunction vector.

The same procedure can be used to write the partial derivative in the x direction in terms of a matrix equation. By comparing Equation 34 (Laplacian) with Equation 35 (partial w.r.t. x), one can see that the equation for the partial derivative, is a simplified version of the Laplacian operator. Therefore, the resulting matrix for the partial derivative w.r.t x should be the same as the A matrix with less diagonal elements and some with –1 instead of 1, and it should be divided by 2δ instead of δ^2. The two matrix operations which represent the partial w.r.t x and y, respectively, are shown below as Equation 38 and Equation 39, respectively.

$$\frac{1}{2\delta}\begin{pmatrix}
 & & & 1 & & & & & & & & -1 & & & & \\
 & & & & 1 & & & & & & & & -1 & & & \\
 & & & & & 1 & & & & & & & & -1 & & \\
 & & & & & & 1 & & & & & & & & -1 & \\
-1 & & & & & & & 1 & & & & & & & & \\
 & -1 & & & & & & & 1 & & & & & & & \\
 & & -1 & & & & & & & 1 & & & & & & \\
 & & & -1 & & & & & & & 1 & & & & & \\
 & & & & -1 & & & & & & & 1 & & & & \\
 & & & & & -1 & & & & & & & 1 & & & \\
 & & & & & & -1 & & & & & & & 1 & & \\
 & & & & & & & -1 & & & & & & & 1 & \\
1 & & & & & & & & -1 & & & & & & & \\
 & 1 & & & & & & & & -1 & & & & & & \\
 & & 1 & & & & & & & & -1 & & & & & \\
 & & & 1 & & & & & & & & -1 & & & &
\end{pmatrix}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix} = \frac{\partial}{\partial x}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix}$$

Equation 38

$$\frac{1}{2\delta}\begin{pmatrix}
 & 1 & & -1 & & & & & & & & & & & & \\
-1 & & 1 & & & & & & & & & & & & & \\
 & -1 & & 1 & & & & & & & & & & & & \\
1 & & -1 & & & & & & & & & & & & & \\
 & & & & 1 & & -1 & & & & & & & & & \\
 & & & -1 & & 1 & & & & & & & & & & \\
 & & & & -1 & & 1 & & & & & & & & & \\
 & & & 1 & & -1 & & & & & & & & & & \\
 & & & & & & & 1 & & -1 & & & & & & \\
 & & & & & & -1 & & 1 & & & & & & & \\
 & & & & & & & -1 & & 1 & & & & & & \\
 & & & & & & 1 & & -1 & & & & & & & \\
 & & & & & & & & & & 1 & & -1 & & & \\
 & & & & & & & & & -1 & & 1 & & & & \\
 & & & & & & & & & & -1 & & 1 & & & \\
 & & & & & & & & & 1 & & -1 & & & &
\end{pmatrix}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix} = \frac{\partial}{\partial y}\begin{pmatrix}\psi_{11}\\ \psi_{12}\\ \psi_{13}\\ \psi_{14}\\ \psi_{21}\\ \psi_{22}\\ \psi_{23}\\ \psi_{24}\\ \psi_{31}\\ \psi_{32}\\ \psi_{33}\\ \psi_{34}\\ \psi_{41}\\ \psi_{42}\\ \psi_{43}\\ \psi_{44}\end{pmatrix}$$

Equation 39

The large, sparse matrices can be called B and C, which perform the operation of partial derivative w.r.t x and partial derivative w.r.t y, respectively.

Using these matrices, the governing equation of motion for the evolution of the fluid vorticity and streamfunction, Equation 30 and Equation 31, can be rewritten as

$$A\bar{\psi} = \bar{\omega}$$

Equation 40

$$\frac{\partial \bar{\omega}}{\partial t} = C\bar{\psi}B\bar{\omega} - B\bar{\psi}C\bar{\omega} + \upsilon A\bar{\omega}$$

Equation 41

A multitude of methods can be used to solve the elliptic problem (Equation 40) for the streamfunction given a vorticity and a standard 2^{nd} or 4^{th} stepping scheme (ode23 or ode45) can be used to advance the advection-diffusion equation (Equation 41) into the future. Once the new value of vorticity is found, the streamfunction can be updated and the process can be repeated.

Elliptic Solve Using Fast Fourier Transform

Once the vorticity is known, the elliptic equation must be solved for the streamfunction. This can be done using a Fourier Transform very efficiently. Recall that a Fourier transform is defined as

$$F(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-ikx} f(x)\, dx$$

Equation 42

The power of the Fourier transform lies in its relation between the function and derivatives of itself. Namely,

$$\hat{F}^{(n)} = (ik)^n \hat{F}$$

Equation 43

Equation 43 shows that the Fourier transform of a derivative of power n of a function is simply the Fourier transform of the original function multiplied by a factor of $(ik)^n$, where n is the number of derivatives. For this problem, the 2-dimensional Fourier transform must be applied since there are derivatives in both the x and y directions. We can rewrite the elliptic equation as

$$\psi_{xx} + \psi_{yy} = \omega$$

By denoting the Fourier transform in x as ^ and the Fourier transform in y as ~, the elliptic equation can first be transformed in x. Note that this does not affect the derivative in y.

$$\hat{\psi}_{xx} + \hat{\psi}_{yy} = \hat{\omega}$$

$$(ik_x)^2 \hat{\psi} + \hat{\psi}_{yy} = \hat{\omega}$$

Now transforming in y yields

$$(ik_x)^2 \tilde{\hat{\psi}} + \tilde{\hat{\psi}}_{yy} = \tilde{\hat{\omega}}$$

$$(ik_x)^2 \tilde{\hat{\psi}} + (ik_y)^2 \tilde{\hat{\psi}} = \tilde{\hat{\omega}}$$

$$-k_x^{\,2}\tilde{\tilde{\psi}} - k_y^{\,2}\tilde{\tilde{\psi}} = \tilde{\tilde{\omega}}$$

This can now be solved for the stream function which is transformed in both the x and y direction. This yields

$$\tilde{\tilde{\psi}} = \frac{-\tilde{\tilde{\omega}}}{k_x^{\,2} + k_y^{\,2}}$$

Equation 44

This expression can then be transformed back into the spatial domain using the inverse 2-dimensional Fourier transform.

This method is desired since it allows for a solution of the stream function in $O\left(n\log\left(n\right)\right)$ operations.

Appendix D: Code User's Manual

Only 4 files are designed to be opened/accessed by the user. These are main.m, user_preferences.m, view_movie.m, and view_subplot.m. The first two are used to drive the simulation and the last two are used for data post-processing.

The simulation can be run by simply running the file main.m. This then prompts the user for input options as shown below

```
Use saved settings or enter settings manually?

  1 = Saved
  2 = Custom
  3 = Demonstration
  4 = Help

Please enter selection:
```

The user can then enter in the solution parameters by choosing option 2 and manually entering parameters.

If multiple simulations are being run, it is possible to simply load saved parameters to bypass the many layers of entering parameters manually. To do this, simply open the file user_preferences.m and modify lines 88-104. Explanations for the various options are located in this file.

A simple demonstration of the code can be run by choosing option 3.

Once the simulation has run, it is possible to post-process the data without having to recreate it each time. To view the simulation (using pcolor, surf, or both at the same time), or to plot the vorticity coefficient and processor time required, simply run the file view_movie.m. Special parameters such as rotating the camera and changing the lighting angle can also be set. Old data can also be loaded directly here if it was saved from main.m.

Similarly, still "snap-shots" of the simulation can be produced by running view_subplots.m after data has been created using main.m (or loaded using view_movie.m). Also, the streamfunction can be plotted using this. However, this requires that the FFT method as the solution method and only update at discrete points in the main.m file. An option will then allow the user to specify if the streamfunction should be plotted in main.m. Once this is generated, view_subplots.m can be used to plot the streamfunction evolution.

Appendix E: Problem Statement

AMATH 581 Autumn Quarter 2003

Homework 2: Vorticity-Streamfunction Equations

The time evolution of the vorticity $\omega(x,y,t)$ and streamfunction $\psi(x,y,t)$ are given by the governing equations:

$$\omega_t + [\psi,\omega] = \nu\nabla^2\omega \tag{1}$$

where $[\psi,\omega] = \psi_x\omega_y - \psi_y\omega_x$, $\nabla^2 = \partial_x^2 + \partial_y^2$, and the streamfunction satisfies

$$\nabla^2\psi = \omega \tag{2}$$

Initial Conditions: Assume a Gaussian shaped mound of initial vorticity for $\omega(x,y,0)$. In particular, assume that the vorticity is elliptical with a ratio of 4:1 or more between the width of the Gaussian in the x- and y-directions. I'll let you pick the initial amplitude (one is always a good start).

Diffusion: In most applications, the diffusion is a small parameter. This fact helps the numerical stability considerably. Here, take $\mu = 0.001$.

Boundary Conditions: Assume periodic boundary conditions for both vorticity and streamfunction. Also, I'll let you experiment with the size of your domain. One of the restrictions is that the initial Gaussian lump of vorticity should be well-contained within your spatial domains.

Numerical Integration Procedure: Discretize (2nd order) the vorticity equation and use ODE23 to step forward in time.

(a) Solve these equations where for the streamline ($\nabla^2\psi = \omega$) use a Fast Fourier Transform.

(b) Solve these equations where for the streamline ($\nabla^2\psi = \omega$) use the following methods:

- A/b
- LU decomposition
- BICGSTAB
- GMRES

Compare all of these methods with your FFT routine developed in part (a) (check out the CPUTIME command for MATLAB). In particular, keep track of the computationl speed of each method. Also, for BICGSTAB and GMRES, for the first few times solving the streamfunction equations, keep track of the residual and number of iterations needed to converge to the solution. Note that you should adjust the tolerance settings in BICGSTAB and GMRES to be consistent with your accuracy in the time-stepping. Experiment with the tolerance to see how much more quickly these iteration schemes converge.

(c) Try out these initial conditions with your favorite/fastest solver on the streamfunction equations.

- Two oppositely "charged" Gaussian vorticies next to each other, i.e. one with positive amplitude, the other with negative amplitude.
- Two same "charged" Gaussian vorticies next to each other.
- Two pairs of oppositely "charged" vorticies which can be made to collide with each other.
- A random assortment (in position, strength, charge, ellipticity, etc.) of vorticies on the periodic domain. Try 10-15 vorticies and watch what happens.

(d) Make a 2-D movie of the dynamics. Color and coolness are key here. (MATLAB command: movie, getframe). I would very much like to see everyone's movies.